Garbage Collection in Distributed Systems

A Thesis
submitted for the

DEGREE OF DOCTOR OF PHILOSOPHY

in the

UNIVERSITY OF NEWCASTLE UPON TYNE

by

SIMON ROBERT WISEMAN

Royal Signals and Radar Establishment
St. Andrews Road
Malvern
Worcestershire

November 1988

Abstract

The provision of system-wide heap storage has a number of advantages. However, when the technique is applied to distributed systems automatically recovering inaccessible variables becomes a serious problem. This thesis presents a survey of such garbage collection techniques but finds that no existing algorithm is entirely suitable. A new, general purpose algorithm is developed and presented which allows individual systems to garbage collect largely independently. The effects of these garbage collections are combined, using recursively structured control mechanisms, to achieve garbage collection of the entire heap with the minimum of overheads. Experimental results show that new algorithm recovers most inaccessible variables more quickly than a straightforward garbage collection, giving an improved memory utilisation.

Contents

1. Introduction

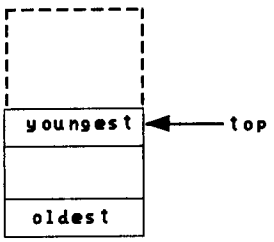
1.1 Stacks, Pools and Heap Storage

Implementations of modern high level programming languages provide the programmer with a variety of automatic storage management facilities. Essentially these provide mechanisms for allocating new variables and for recovering the space occupied by unused variables. They are designed to hide low level details, such as memory address allocation, in order to make programs easier to write and maintain and to enhance portability.

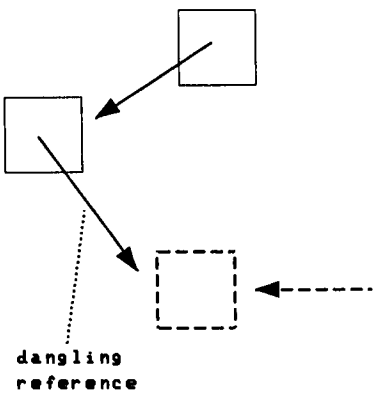
Three types of storage management facility can be distinguished; stacks, pools and heap stores. They each have different properties, illustrated in figure 1.1a, which make them suitable for different tasks. The stack is widely used to allocate space for variables in block structured languages since allocation and deallocation are extremely efficient. Pools and heaps are often provided as a means of generating variables which have longer lifetimes than that of the scope of a program's identifiers.

Using a stack, variables may only be deallocated in the reverse order to which they were allocated, which corresponds directly to the scoping of identifiers in a block structured language. However, it is not possible, using stack storage, to create a variable and refer to this after exit from the scope in which it was created, which is desirable when manipulating linked data structures, such as linked lists and trees.

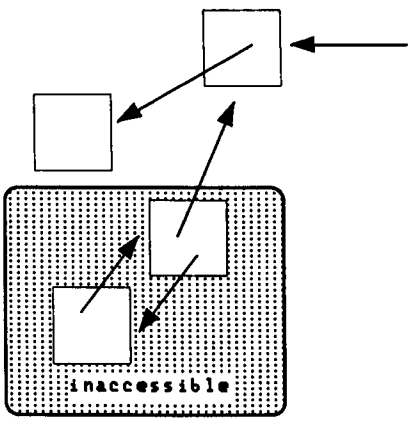
Variables allocated in a pool or heap store can have lifetimes longer than the scope of the program's identifiers. Thus variables can be referenced after the program exits from the scope in which they were created. For example, a procedure could allocate a new variable in the heap and link this into an existing list. The new variable would remain allocated when the procedure exits because its lifetime extends beyond the scope of the procedure's body.



Stacks are cheap to maintain, but are difficult to use for linked data structures.



Variables in a pool are explicitly deallocated which can give rise to dangling references.



Variables in a heap are recovered automatically when they become inaccessible.

Fig1.1a: Stacks, Pools and Heaps

Heap storage and pools can be distinguished by the way in which storage is recovered. Variables in pools are deallocated explicitly by the program. For example, in Pascal the procedure dispose is used to deallocate the variable referred to by a pointer variable. In contrast, recovering variables in heap stores is performed automatically, either by the language's run time support system, the operating system or by the hardware.

Explicit deallocation is potentially more efficient than automatic deallocation, in terms of CPU time and memory utilisation. However the problem with explicit deallocation of variables is that the variable may still be referenced elsewhere. Subsequent use of this reference would lead to serious, and difficult to trace, errors as the released storage may since have been reused to allocate new variables. This is known as the dangling reference problem.

The burden of keeping track of references in a pool is therefore placed on the programmer. However, with all but the simplest linked data structures, this burden becomes intolerable. Programs are extremely difficult to debug if mistakes are made.

The use of tombstones [Lomet75] would prevent the use of dangling references if they are formed. This involves marking the variable in such a way that all accesses can distinguish between an allocated variable and a deallocated one. However, this is really just deferring the problem, since it is just as difficult to remove the tombstones safely as it is the variables.

Heap storage is distinguished from pools in that no explicit deallocation mechanism is provided. A program may allocate a new variable at any time and references to it may be copied, distributed and stored in complex linked data structures. However the program or programmer does not have to keep track of these references in order to detect when the variable is no longer of any use. Deallocation of the variable is performed automatically, using a technique called garbage collection, which is described in the next section. The garbage collector will recover a variable only if it can determine that the variable will never be accessed in the future.

Heaps and pools occur in programming languages under a variety of guises, as shown by the examples in figure 1.1b. The use of the heap in Algol68 is quite explicit, whereas in Simula it is used when processes are created and in LISP heap storage is used to store lists. In Pascal and Ada explicit pools are provided. Object oriented systems use either heap storage or pools [Gorlen87] and special purpose heap stores are found in applicative systems [Clarke et.al.80].

```
PROC add = ( REF LIST l, INT d ) VOID:
BEGIN
  l := HEAP LIST := ( l, d )
END

f[ x ] = [ atom[ x ] → x; T → cons[ f[car[x]];f[cdr[x]] ] ]

element ford;
activity person( age ); integer age;
begin .... end;
ford := new person( 42 );
```

Fig1.1b: A heap is used explicitly in Algol68, used to implement lists in LISP and to create processes in Simula.

Heap storage is usually provided by a programming language's run time support environment, however there are some computer architectures which provide heap storage at a much lower level. Capability computers, which provide general purpose computing, and LISP engines both offer heap storage at the instruction set level, often with microcoded heap management routines.

1.2 Garbage Collection

Garbage collection is the means whereby variables that are no longer needed are recovered, so that the memory they occupy can be reused for the allocation of new variables. A variable is only recovered if no program can, at any time in the future, access it. In this way the dangling reference problem found with pools is avoided. The technique was first described by [McCarthy60], but the term "garbage collection" does not appear until [Schorr&Waite67].

The only way a general purpose garbage collector can decide that a variable will never again be accessed is by determining that no reference to it remains accessible to any program. The assumption here is that references to variables can be distinguished from other data and cannot be spontaneously created. That is, a new reference can only be made by creating a new variable or copying an existing reference.

A problem with relying on garbage collection to recover unwanted variables is that references to them may inadvertently be kept accessible. Errors of this form are quite difficult to track down and can cause a severe degradation in memory utilisation.

A typical example, illustrated in figure 1.2, is an implementation of a stack of references which uses an array. Popping an item from the stack is implemented by decrementing a stack pointer. Unfortunately this does not remove the reference from the array, and so the unstacked variable remains accessible even when it is no longer wanted. A better implementation would of course simply clear the word on the stack as it is popped.

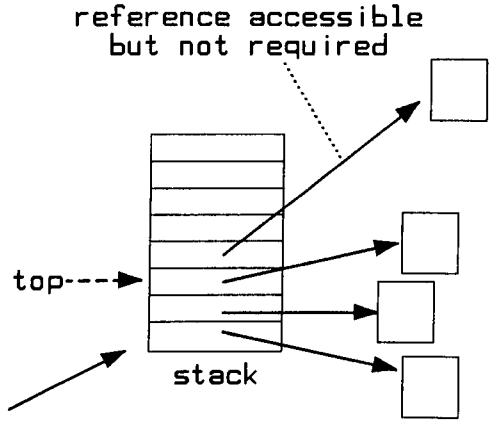


Fig1.2: Care is needed to ensure that unwanted references are made inaccessible, because garbage collection only recovers inaccessible variables.

There are many techniques for locating inaccessible variables, the so called garbage. There are the usual space versus time tradeoffs between the different techniques, but they also differ in the way they function. Some require that normal processing is suspended while the garbage collection is performed, while others allow it to continue. Some algorithms inherently compact the free storage while others leave the recovered storage in place, scattered about memory.

Regardless of the garbage collection technique employed, it is necessary to identify references to variables and distinguish them from scalar data. This can be complex, involving searches of symbol tables generated by compilers, as in the case of Algol68, or be given simply by the state of a single bit, as in LISP.

When support for garbage collection is built into the hardware, as in capability computers and LISP engines, the distinction between references and scalar data is maintained by the hardware. This may be done tagging individual words within a variable with an extra bit which indicates whether the word currently holds a reference or a scalar value. This technique is used in the Flex capability computer [Foster et.al.82] and in LISP engines, such as the Symbolics [Moon85]. An alternative is to tag the whole variable. Thus variables may either hold only references or only scalar data. This technique has been employed in many systems, such as the Plessey PP250 [England75] and Cambridge CAP computer [Needham&Walker77]. An alternative, used in later versions of Intel's iAPX432 [Tyner81] and the HIP processor [Menu et.al.87], is to divide variables into two parts, one that contains only references and one that contains only scalar data.

There are essentially two ways of finding inaccessible variables, the scanning technique [McCarthy60] and the reference counting technique [Collins60]. There are many variants of these two approaches, and they are often used in conjunction or combination with each other. Scanning garbage collectors work by finding all accessible variables and then deducing that all other variables are inaccessible. This is done using a recursive scan through the accessible heap structure. The reference counting technique maintains a count of the number of references that refer to each variable. If the reference count of a variable drops to zero, the variable is known to be inaccessible garbage.

However, garbage collection is only one part of a heap's storage management system. Storage management not only covers allocation and recovery of variables, but addresses compaction of free store, imposing budgets on the use of memory resources and perhaps the optimisation of paged memory usage. These topics cannot be considered in isolation, for example allocation is much simpler if the free memory is contiguous but compaction is then more of a problem.

So the effectiveness of a garbage collector cannot be measured simply in terms of the memory bandwidth required to perform garbage collection. Many other factors need to be taken into account, such as its effect on the normal operation of the computer and the performance of variable allocation. In particular some extremely memory efficient garbage collectors are useless in certain applications because of the disruption they cause to normal processing.

1.3 The Advantages of a System-Wide Heap

In most computer systems, each program has its own address space. Variables allocated by one program cannot be referenced by another. If a reference to a variable of one program is somehow passed to another, perhaps by a message passing system, it changes its meaning as it will then be interpreted in the wrong context.

As discussed earlier, the advantage of using heap storage rather than stack storage, is that variables have lifetimes longer than that of the scope in which they were created. However in almost all systems the lifetime of a variable does not extend beyond the lifetime of the program which created it.

If a system provides a single system-wide heap store, which is shared by all programs, there are considerable gains to be made. The single address space allows programs to be composed into systems in a uniform way. This makes them easier to implement and maintain, as has been demonstrated with the Flex capability computer, [Stanley85a] and [Stanley85b], and is proposed under the persistent storage approach, [Atkinson&Morrison85] and [Morrison et.al.87]. This is due to a number of factors.

First, and perhaps most important, is that variables can be allocated whose lifetime is that of the system, rather than that of the program that created them. In this way a program can construct complex structures and return them as results, without having to resort to intermediate files on backing store, which is the case in most systems. For example the interface between the front and back ends of a compiler is usually a file on disc. If a system-wide heap store is available, the front end can construct a tree structure as an intermediate result and pass this on to the back end.

In this way the distinction between programs and procedures becomes blurred. Programs can take parameters and return results using the heap, in exactly the same way as procedures. It is easier to construct and maintain large systems in this uniform procedural way than to use many independent programs interacting through files or channels for flat scalar messages. Interfaces between modules can be expressed in terms of complex data structures, rather than flat files. Programs can be incorporated in others, just like procedures can be reused in high level languages, giving greater software reuse.

Secondly, programs executing in parallel can communicate through the heap store, sharing variables and passing structured values rather than just scalar data, as is the case with simple message passing systems. In systems which allow limited amounts of shared store between programs, the programmer must decide where to allocate variables, according to whether they are to be shared or not. Often this decision cannot be made until run time, so more data than is necessary is made shared. Also the memory may be mapped into different places in the address spaces of the programs, which makes it difficult to pass references (addresses) between the programs.

Thirdly, sharing the heap store gives better utilisation of the computer's memory. If each program has its own heap, it is necessary to allocate to each of them enough memory to satisfy their peak demand. By sharing the heap, the amount of memory that needs to be allocated is the maximum required at any one time, which may constitute a considerable saving.

Also, the flexible communication between sub-systems which is achieved by using a system-wide heap is an essential part of the technique of object oriented programming [Bhaskar83]. An object is an abstract representation of some data. It is defined purely in terms of the behaviour of the operations that act upon it. This ensures the user of an object cannot rely on some particular implementation detail which the implementor later changes.

However, the use of a system-wide heap is not without its problems. In particular garbage collection must involve all programs in the system. Thus if one program requires large amounts of garbage collection, programs that need relatively little are penalised. Also, if one program consumes a large amount of the shared heap, either deliberately or accidentally, it may cause others to fail because their store requirements cannot be satisfied.

1.4 System-Wide Heap Stores in Distributed Systems

The idea of a system-wide heap can be extended to distributed systems. Here the heap is shared by all the programs in the whole system. Thus a reference to a variable created on one computer can be passed to another computer and yet still remain valid. However, it may not be possible to use the reference to access the variable directly. This would depend on the system's semantics of remote variables. The reference may always of course be passed back to the originating computer, perhaps as a parameter to a remote procedure call [White76], where it can be used to access the variable.

The advantage of providing a system-wide heap in distributed systems is that they can be constructed in a uniform way, using procedural interfaces with structured parameters and results. Existing systems are more readily combined and new systems can be designed, implemented and tested independently of the configuration of the system.

As the review in chapter two reveals, some garbage collection strategies do not guarantee to recover all inaccessible variables. When the heap is contained within one program this may not present a serious problem, since all variables created by the program are immediately recovered when it terminates. However it is most important in distributed systems where the heap is spread across many nodes of a network, because the lifetime of the garbage is that of the node containing it. If the garbage collector was not effective, the heap would slowly but surely fill up with inaccessible variables, until the nodes containing the garbage were restarted.

By extending the system-wide heap store across a distributed system, the problems of garbage collection become considerable. Using conventional techniques the time required to garbage collect a distributed system will be many times that needed for individual systems. This is because of relatively large communication delays, even though garbage collection can be processed in parallel by each individual system.

If normal processing must be suspended while garbage collection takes place, extremely large pauses in execution will occur, which is unlikely to be acceptable to any application. Even if normal processing can proceed in parallel with garbage collection, the number of new variables generated between garbage collections will be proportionally greater. This means systems will have to provide much larger memories for the heap store in order to satisfy demands for new variables while the garbage is being recovered. Such poor utilisation of memory would make distributed heaps far too expensive, probably outweighing the advantages they offer.

1.5 Contribution of the Thesis

This thesis tackles the major problem of providing garbage collection in distributed systems with system-wide, single address space heap stores. The scale of distributed systems is such that, using conventional garbage collection algorithms, either unacceptably long pauses in execution would occur, or poor utilisation of memory would make the systems excessively expensive.

The main contribution of the thesis is to show how the traditional technique of divide and conquer can be used to limit the time required for garbage collection. This is achieved by recursively structuring the distributed heap. However the thesis shows how the recursion can be eliminated, providing a practical approach to garbage collection of large distributed systems. The algorithm presented is shown to exhibit the essential properties of safety, effectiveness and timeliness.

In spite of the recursive structure of the heap, a single address space is presented to the users. The internal structure is relevant only to the administration of the heap store. However, choice of this structure is most important, since, as it will be shown, the efficiency of the garbage collection depends greatly on the locality of references within the recursive structure.

The recursive structuring principle is an effective way of combining existing systems together. The distributed heap store, because of its single address space, allows programs in the different systems to communicate and share data without the need for extensive rewriting. An important claim of this thesis is that garbage collection of such large distributed heaps can be performed effectively. The overhead imposed on the individual systems is minimal and is in relation to their involvement in the distributed data structures.

Using the new garbage collection algorithm that is developed in this thesis, systems with different individual approaches to garbage collection can be combined. This is particularly important since most systems are made to optimise the performance of a particular environment, such as LISP. This thesis investigates the various techniques for garbage collection and shows how these can be incorporated into a recursively structured system.

Another important contribution made by the thesis is to extend recursive structuring into the heap store held within single computers. Logical partitions within a heap can be used to control the amount of memory allocated by a program, and can be used to limit the effects of one program's garbage collection on another.

The thesis also addresses the problem of operation in a faulty environment, which is essential when considering distributed systems. The most difficult problem arises when communication with other nodes is lost. This is because it is often impossible to distinguish between temporary partitioning of the network and the crashing of some nodes. If nodes crash it can be assumed that all references they contained have been destroyed, whereas partitioning may mend and references once more become usable. It is shown that relatively straightforward techniques can be applied to solve these problems.

The thesis therefore makes significant contributions to the practical development of systems with distributed, single address space heap stores. This is of particular relevance with regards the implementation of distributed capability systems. In addition, it is planned to use the algorithm to garbage collect logical and physical areas of store in the SMITE multiprocessor capability computer, [Harrold&Wiseman88] and [Wiseman&Field-Richards88], and the SMITE structured backing store [Wiseman88].

1.6 Organisation of the Thesis

Chapter two presents a survey of existing garbage collection techniques found in the literature. This covers the two broad styles of garbage collection, namely reference counting and scanning. A critique is presented which compares the suitability of each algorithm against the essential requirements of garbage collection in a single address space distributed heap store. While most of these algorithms are intended for use in single processor systems, some have been developed for use in distributed systems. Unfortunately these are found to suffer from drawbacks which mean they are not entirely suitable for general purpose distributed heaps.

In chapter three, the recursively structured heap is described in detail and a recursive garbage collection algorithm is presented. This is then refined to a parallel recursive algorithm. This algorithm is still recursive but performs garbage collection by launching parallel processes to work on inner levels of the heap. Next the recursion is eliminated to yield a non-recursive parallel algorithm and the effects of parallel computations are considered. This shows how the normal computation must interact with the garbage collector to ensure the heap is correctly managed. Finally a formal specification of the garbage collector is given. Part of this is refined to code to illustrate the techniques that can be applied to produce an implementation which is proven correct.

The practical implementation of the algorithm is considered in chapter four. Several methods are explored which correspond to various distributed system architectures. The chapter considers the practical problems arising from the distributed environment, including termination detection and fault tolerance.

An analysis of the algorithm's performance is made in chapter five. Equations are developed which show the worst case and typical memory utilisation achieved when using the new algorithm. These are used to compare it against a simple straightforward garbage collection of the entire heap. The results of measurements taken of a real system are presented to give an indication of the typical values of the parameters involved in the equations.

The thesis concludes by summarising the new algorithm and discussing its applicability to real systems. The limitations of the algorithm are also discussed, and avenues for future work are explored.

2. Garbage Collection Techniques

2.1 Requirements

Many garbage collection algorithms are reported in the literature and they fall broadly into two styles, Reference Counting and Scanning. A comprehensive review of garbage collection algorithms is presented by [Cohen81]. This chapter presents an alternative review, of these algorithms and of more recent work, which evaluates their suitability for use in garbage collecting a single address space, distributed heap store. First, however, some criteria for selecting an algorithm must be developed.

2.1.1 Essential Criteria for Garbage Collectors of Distributed Heaps

An algorithm that recovers variables which are still accessible is clearly unacceptable, as is one which corrupts the accessible data structure in the heap store. However it is acceptable for physical storage to be compacted. That is variables may be moved as long as all accessible references to them are updated to reflect their new location. These requirements can be summarised as the following criterion of safety:

C1: The actions of the garbage collector leave the logical structure accessible to programs unchanged.

The garbage collector must be able to recover garbage regardless of the actions of the programs using the heap store. Even a pathological program must not prevent garbage from eventually being recovered, because this could affect all programs using the heap. The complexity of the structure in the heap store may affect the amount of workspace required by the algorithm. It must be possible to establish a bound on the size of this, to ensure that it can never be exhausted, so that garbage can always be recovered. This leads to the second criterion, effectiveness:

C2: The garbage collector must recover any inaccessible variable in bounded time and with bounded workspace.

Some algorithms are tailored to particular applications, such as a heap for LISP programs. However a distributed heap is likely to be used by a wide variety of applications written in a variety of styles, using various programming languages. The garbage collector must therefore be somewhat general purpose in nature. In particular it should allow for variables of varying sizes, unlike the fixed sized variables of LISP, that are capable of containing mixtures of both reference and scalar data. Also it must cater for programs that create cyclic structures in an arbitrary fashion, unlike functional languages which produce cyclic structures only in very particular ways. Thus the criterion of generality is:

C3: The garbage collector must cater for applications which require that the variables in the heap store are of arbitrary size, may contain both reference and scalar data and any assignment to a reference may form a cyclic structure of arbitrary complexity.

Garbage collectors which cause lengthy pauses in execution, because they can only operate while no normal processing occurs, are not acceptable for distributed heap stores. Such garbage collectors are acceptable for some applications, such as single user workstations where the pauses only cause an occasional minor disturbance to human input. However a distributed heap is likely to be very large which, coupled with communication delays, will cause the pauses to be unacceptably long.

It must be accepted that all garbage collectors cause some disturbance to the execution of programs. However it is desirable that the pauses in execution caused by these interactions should be small, so that the response time of a system is acceptable. Further, the length of the pauses should be bounded so that some real time response can be guaranteed. A garbage collector for a distributed heap must therefore be suitably unobtrusive:

C4: The time taken executing critical sections between the garbage collector and normal processing must be bounded and small.

The first two criteria are generalisations of the correctness criteria given for the algorithm of [Dijkstra et al. 78]. These state that the garbage collector must recover all inaccessible variables, and no others. The last two criteria state that, to be of use in a distributed system, the garbage collector must be general purpose and not interfere with the system's normal computation.

2.1.2 Method of Comparison

All the algorithms surveyed satisfy the first essential criterion, that is they do not alter the logical structure in the heap. Most satisfy the second criterion, in that they use bounded time and workspace. However a few cannot guarantee to collect all garbage in bounded time and for others, while the workspace required is bounded, an excessive amount is needed.

Different algorithms that do satisfy all the essential requirements will represent a tradeoff between several factors. Unfortunately the precise relationship will be application dependent. However, since the system is intended to be general purpose, the aim should be to achieve reasonable results across a range of typical applications. This is possible if traits which are exhibited by programs in general are exploited.

The tradeoffs will be between features such as real time response, size of workspace required by the garbage collector, synchronisation overheads, cost of any special purpose hardware and memory utilisation. The latter is a particularly important measure. It is given by the minimum amount of store in a heap required to ensure that an application can run continuously without exhausting the available free store. This depends on the rate at which inaccessible variables can be recovered by the garbage collector. The more rapid the recovery for a given amount of overhead, the smaller the required memory and hence the cheaper the system. Conversely it can be thought of as a measure of the 'size' of the largest program that can be run without pauses on a given system.

A garbage collector for a distributed heap should exhibit additional properties not required for heaps in single computers. It should be tolerant of the communications failures which will inevitably arise and cope with node crashes which destroy part of the heap and with network partitioning. Also the distributed environment is likely to bring together a variety of systems which provide their own local garbage collectors. These may be optimised for particular environments, such as LISP, and the distributed garbage collector should allow these to be integrated into the distributed system. In particular, the provision of distributed garbage collection should not unduly interfere with the internal workings of these systems and should allow them to continue benefitting from their specialised garbage collectors as much as possible.

2.2 Reference Counting Algorithms

The reference count method, first introduced by [Collins60], attempts to detect when a variable becomes inaccessible. For each variable, a count is maintained of the number of references to it that are stored in the heap. If the count ever drops to zero, then the variable has become inaccessible and so its storage can be freed. This is shown in figure 2.2a.

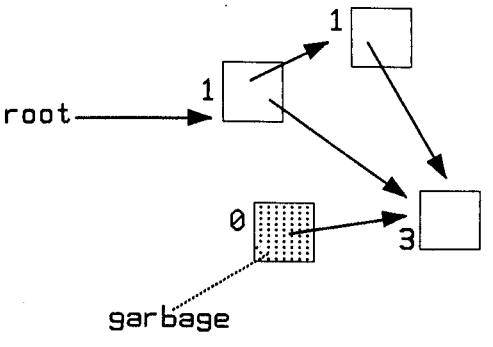


Fig2.2a: The number of references to each variable is recorded. If this drops to zero the variable can be recovered.

Before the storage can be used again for allocating new variables, it must be scanned for references to other variables. This is so that the reference counts of the variables they refer to can be decremented, which may recursively cause further variables to be freed. This searching can either be done when the variable is first released or when it is reused and may be associated clearing the space to zero.

The count must be maintained when references are created and destroyed. When a variable is created, a reference to it is returned to the creator, therefore the reference count is initially set to one. Further references are created simply by copying, so it is necessary to check all memory writes to see if the data being written is a reference. If so, it is necessary to increment the reference count of the variable referred to by the data.

A reference is destroyed if it is overwritten, in which case the reference count of the variable that it refers to must be decremented. If references can be stored anywhere in any variable, it is necessary to check whether a location contains a reference before writing to it. Note that the incrementing must occur before the decrementing to ensure that a variable is not prematurely recovered if the last reference to it is overwritten with itself.

Reference counting does not detect all inaccessible variables. This is because a variable can become inaccessible by the destruction of a reference which is not the last reference to the variable. The simplest example is where a variable contains a reference to itself and is referred to by one other reference stored in an accessible variable. If the latter variable is destroyed, then the variable's reference count decreases from two to one. The variable is therefore not freed even though it has become inaccessible. This is in general true for an arbitrarily complex cyclic structure, as in the example shown in figure 2.2b.

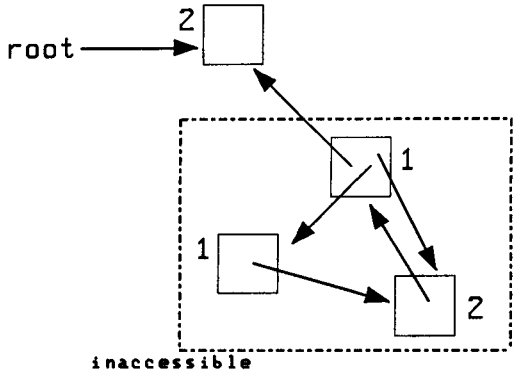


Fig2.2b: Variables which form an inaccessible cyclic structure are not recovered because their reference counts do not drop to zero.

2.2.1 Recovering Inaccessible Cyclic Structures

The inability to detect inaccessible cyclic structures is the greatest disadvantage of reference counting. This is especially so when the main memory of a computer is organised as a heap, because it is quite possible that large cyclic structures will be created. For example, the data structures of the scheduler will contain references to all the processes in the computer. These processes will have references to the synchronisation primitives supplied by the scheduler, which themselves contain references to the scheduler data structures, thus forming a cyclic structure. The structure will become inaccessible if, for example, two processes deadlock by claiming semaphores which only the other references.

The problem of inaccessible cyclic structures can be overcome, either by detecting when a circularity is produced and then not incrementing the reference count, or by ignoring the problem and using another technique to recover them.

The latter solution is proposed by [Deutsch&Bobrow76] and [Christopher84]. In these hybrid schemes, reference counting is used to recover garbage, except for inaccessible cyclic structures, until there is no free storage left. Then a scanning technique, see section 2.3, is used to recover any inaccessible cyclic structures. A hybrid scheme is also used in the Cedar programming environment [Swinehart et.al.86]. Here programs are designed to explicitly break cycles if they can determine that the data structures are no longer needed in order to improve efficiency.

The former solution was proposed by [Weizenbaum62] and [Weizenbaum63] though, as pointed out by [McBeth63], detecting when a circular structure is formed involves a search of potentially the entire memory. There are special cases where it is known when circularities are produced, such as the use of the Y-combinator in combinator based systems [Turner79]. Systems which take advantage of this are described in [Friedman&Wise79], [Hughes85], [Brownbridge84] and [Brownbridge85]. However, these are not applicable to general purpose heap stores, and a study by [Watson86] concludes that these techniques are very complex and in practice cause the garbage collector and computation to interact closely.

Another method is proposed by [Bobrow80]. In this, all the variables of a circular structure are treated as a single group for deallocation purposes. A reference count to the group as a whole is maintained, but no counts within a group are kept. This scheme is not particularly suitable for a general heap store as memory usage is not divided into convenient groups and cyclic structures can be quite large.

A radical solution to the problem of recovering cyclic structures is proposed by [Dennis74]. Here a programming language is described which has been developed to have clean semantics and to specifically avoid the use of cycles in its implementation. The obvious drawback of this approach is that is does not cater for standard programming languages.

2.2.2 Recursively Releasing Storage

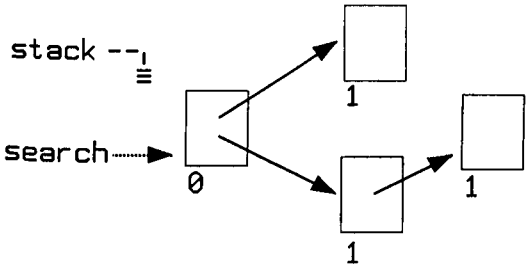
The procedure for freeing a variable is recursive, since a freed variable may contain references to other variables which consequently become free. Freed variables must therefore be searched for references, so the reference counts of the variables they refer to can be decremented. The search may either be done when the variable is first freed or it may be deferred until the storage occupied by the variable is reused.

With the former approach, variables are searched for further references when they are first freed and a stack of some sort is required to control the recursion. Since the number of variables freed recursively is potentially all the variables in the memory, the size of the stack must be the same as the maximum number of variables that can be allocated, to handle the worse case.

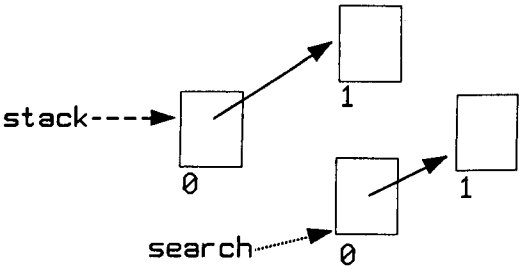
The size of stack required for this is such that using a separate memory for it is prohibitively expensive. The use of a transaction file on disc, as in [Deutsch&Bobrow76], or a virtual memory system would be possible, but would be slow. The stack could be limited to some affordable size as long as stack overflow can be handled and does not happen very often. Finding variables that require scanning, but are not on the stack because of overflow, involves visiting all the variables in memory to find those with a reference count of zero.

A means of avoiding the pauses in computation that can occur while reference counts are recursively decremented is proposed by [Sohi et.al.85]. The LISP oriented architecture they describe uses a separate processor which is dedicated to recursively decrementing the reference counts.

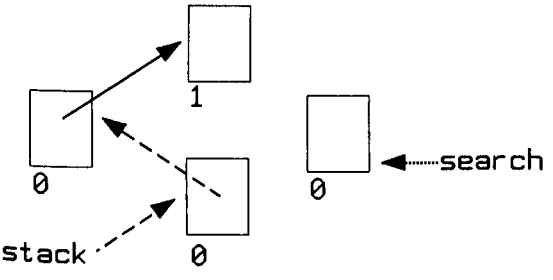
The use of a separate stack can be avoided altogether by utilizing the storage of the variables themselves [Schorr&Waite67]. This is illustrated in figure 2.2c. If a variable is no longer referenced it is scanned for references. If one is found, the variable it refers to has its reference count decremented. If the count of this variable falls to zero, scanning of the first variable is suspended and the new variable is scanned. The location occupied by the reference is used as the link in the chain of variables that have not been completely scanned. When scanning of a variable is completed, the next variable is removed from this chain and the scanning of it continues.



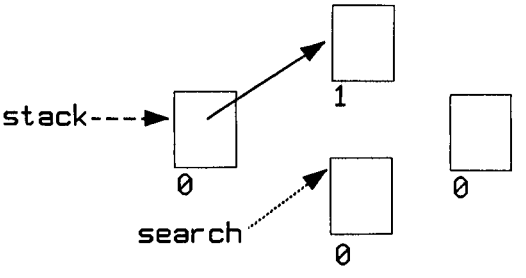
1: The search begins with the freed variable. A reference is found and the variable's reference count is decremented to zero.



2: The position of the search is remembered on the stack and the newly freed variable is searched.



3: When another reference is found, and the variable's reference count is decremented to zero, the search position is stacked again.



4: The stack is unwound when the search reaches the end of the variable.

Fig2.2c: Freed variables can be recursively searched for references by using the storage occupied by the references as a stack.

The location forming the link is, of course, the place that was scanned last. This scheme requires that the end of the variable can be identified, as there is unlikely to be any room to store how much more scanning is required. Alternatively, if the start of a variable is distinguishable, the scans can progress backwards.

The alternative approach of delaying the search of a freed variable until the storage it occupies is reused, obviates the need for the stack [Weizenbaum63]. When a new variable is allocated, its store is cleared to zero (say), and at this point any references it contained will be decremented by the usual reference overwriting mechanism.

By deferring the scanning, inaccessible variables with non-zero reference counts will exist in the heap. Such variables are obviously not available for allocating new variables. This is a potential problem for systems with arbitrary size variables. The free list may not contain a fragment large enough to allocate the new variable, although the store occupied by inaccessible variables which are yet to be recovered could satisfy the request. Thus allocation could be delayed while the free list is cleared in an attempt to recover more store. Hence the technique is most suitable for systems with fixed size variables.

2.2.3 Storing the Reference Count

An obvious requirement for a reference counting system is that a reference count for each variable must be stored somewhere. The reference count field of a variable must be large enough to hold the maximum value that its reference count will ever reach. The worst case occurs when the memory is full of references to the variable. Catering for this possibility is potentially very wasteful of memory, since most variables are referred to by very few references. Two schemes have been proposed by [Deutsch&Bobrow76] to reduce this overhead.

In the first scheme, the reference count field is made much smaller than the maximum required. Whenever the count reaches its maximum it is assumed to be 'infinite' and is never subsequently incremented or decremented. The count can therefore never reach zero so the variable will never be freed. The scanning garbage collector which is used to release inaccessible cyclic structures also releases inaccessible variables which have an infinite reference count. It is hoped that most variables are referenced very few times, so that reference counts becoming infinite is a rare event.

The second scheme is based upon the assumption that the vast majority of reference counts are one. A hash table is used to record the reference count of all variables whose reference count is greater than one. If a variable is not in the hash table then a count of one is implied. However, the problem of hash table space overflow, which is not addressed, makes this approach unattractive for a computer with a heap as its main memory system and for distributed systems.

[Wise&Friedman77] propose the use of a single bit as a reference count. This indicates whether the count is one or greater than one. In addition, a simple cache memory is used to record some of those variables whose reference count is two. This is on the assumption that most variables have a count of one, but that they often increase to two temporarily, during reference manipulation operations. An analysis of LISP programs by [Clark&Green78] supports this view. Wise and Friedman suggest using the mark bit, required for the scanning garbage collector, as the reference count field. It is necessary to clear all the bits before scanning, but the reference count is easily restored afterwards.

A method for recomputing reference counts during scanning garbage collection is proposed by [Wise79]. This would be used to correct those reference counts which have stuck at the maximum, but where the variable is actually referenced by fewer references. In this method the number of references is computed for each variable at a time, so only one access is made to each variable's reference count field. This contrasts with the more obvious technique of incrementing the reference counts of variables whenever a reference is found during scanning, which requires many increments to be performed for each reference count.

2.2.4 Accessing the Reference Count Field

During normal processing many references are created and destroyed, which means many increment and decrement operations are performed on the reference counts. [Deutsch&Bobrow76] propose postponing all increment and decrement actions by storing them in a transaction file. These are then processed by the system some time later, during a slack period.

The number of transactions stored in the file is reduced by ommiting those caused by moving references to or from the computation's local workspace. Empirical studies of LISP systems [Clark&Green77] have shown that the vast majority of reference copying takes place in the local workspace. Hence a considerable saving is made. The references held in the local workspace are counted with a simple scan when the transaction file is processed. Using these techniques [Deutsch&Schiffman84] found that 85% of reference counting operations were eliminated. Improvements suggested by [Suzuki&Terada84] eliminate further operations by not considering references pushed temporarily onto the stack.

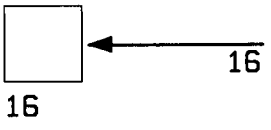
Further techniques for reducing the number of increments and decrements are given by [Barth77], but these are compile time optimizations meant for language run-time systems.

[Wise85] suggests building a special memory interface which contains a processing element dedicated to maintaining reference counts. In this way the overheads of incrementing and decrementing the counts, and of recursively freeing variables, is absorbed by processing in parallel with normal computation. The proposed hardware also performs a scanning garbage collection, when necessary, to recover inaccessible cyclic structures and recompute the reference counts.

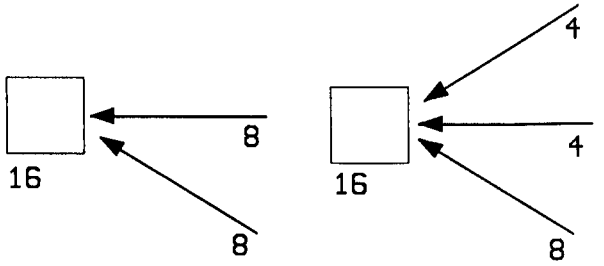
If reference counting is used in a distributed system, care must be taken to ensure that decrements and increments are not made out of order. As is explained in detail in section 4.6, any outstanding requests to increment the reference count must be considered before deallocating a variable. While this can be overcome by ensuring the requests are delivered in the same order that they are sent [Nori79], by using a two way synchronisation protocol or timestamps [Liskov&Ladin86], the overheads imposed can cancel out the benefits of using reference counting [Watson&Watson87].

The solution offered by [Snyder79] does not free a variable as soon as its count reaches zero. Instead a list of such variables is maintained and occasionally the whole machine is stopped and all outstanding increment and decrement requests are processed. Then the list is consulted and any variable with a zero reference count is recovered. Unfortunately this does not work well in a distributed system because of the difficulty, and undesirability, of stopping the whole system and of detecting when all reference count transactions have been processed.

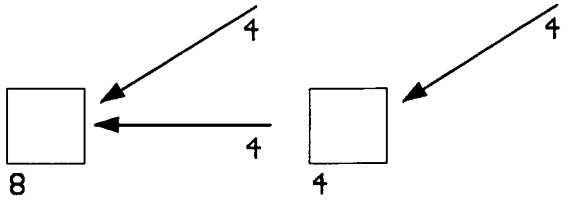
The Weighted Reference Count scheme is a more elegant solution to the problem and is described by [Watson&Watson87]. In this scheme reference count weights are associated with the references as well as the reference counts on the variables. The system maintains the invariant that the sum of the weights of all the references to a variable equals the reference count of the variable. The technique is shown in figure 2.2d. When a variable is allocated, both its reference count and the value in the initial reference are set at maximum. Whenever a reference is copied, its weight is split between the original reference and the copy. Whenever a reference is destroyed the count of the variable is decrease by the value of the weight. In this way the reference count field of a variable is never incremented, and so there is no possibility that the count could reach zero before the variable becomes inaccessible.



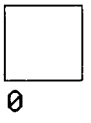
1: When a variable is first allocated it is given the same weight as the new reference.



2: When a reference is copied its weight is split.



3: When a reference is destroyed its weight is subtracted from the variable's weight.



4: When the variable's weight drops to zero it can be recovered.

Fig2.2d: With the Weighted Reference Count scheme a variable's count is never incremented and so it is suitable for use in a distributed system.

The scheme suffers from two drawbacks. Firstly space must be found in the references to store the weights. Secondly some special action must be taken when references with a weight of one are copied. Watson and Watson offer a data compression technique to alleviate the first problem, and suggest fitting a hidden indirection through a new variable to solve the second problem.

2.2.5 Reading Before All Writes

The requirement to read a location before writing to it, in order that the overwriting of references can be detected, is likely to be a serious handicap to performance. This read can, of course, be avoided if it is known that no reference could possibly be stored in that location, though in general this is not so.

This problem is much less serious if scalar and reference data cannot be freely mixed in the same variable. In this case the read check need only be performed when a reference is written to memory. Also, computers with this partitioned type of heap store tend to manipulate references less often than those with the general type.

2.3 Scanning Algorithms

If each allocated variable has a mark bit, and initially all these are clear, then by tracing all the accessible variables and setting their mark bit, it is possible to discover all inaccessible variables. This method, first proposed by [McCarthy60], is recursive.

2.3.1 Simple Scanning

McCarthy's algorithm uses a separate stack to control the recursion. However, to cater for the worst case, the stack would have to be impractically large. A more practical proposal is made by [Hanson77] which suggests the use of a spare location per variable. This is used to link together the variables which have yet to be scanned. The HIP processor, [Menu et.al.87] and [Sanchez et.al.87], uses a microprogrammed version of this algorithm.

A further variation of McCarthy's garbage collector is given by [Baecker72]. This is intended for use in virtual memory systems and has one mark bit per page as well as one per variable. A single recursive scan is made to determine which pages contain accessible variables. Pages that contain only inaccessible variables are then freed along with their page table entry. The advantage of this system is that compaction is unnecessary, but the disadvantage is that pages are not freed until they are completely inaccessible.

To cater for the worst case the stack must have one word per allocated variable. However, it is found in practice that this amount is rarely required [Kurokawa81]. Methods to reduce the amount of space are proposed by [Kurokawa81] but if stack overflow does occur no garbage can be recovered and the system must presumably halt.

2.3.2 Reference Reversal

A method which does not require an auxiliary stack is given by [Schorr&Waite67] and more formally by [Broy&Pepper82]. The algorithm scans a variable looking for references to variables which have not been marked. When one is found the position of the reference is remembered on a list, using the location itself as the link. Scanning then continues in the new variable. When scanning a variable is completed, the list is popped to find the location of the reference which led to the variable. The reference's value is restored and scanning then continues at the location after the reference. This is the same principle as that used with reference counting to recursively release variables, described in section 2.2.2.

When the algorithm returns to continue scanning a variable, it must be able to determine how much more scanning is required before the variable is completely scanned. Since Schorr and Waite are dealing with LISP structures they only require a single bit per node to indicate which word has been scanned; the reversed references actually refer to the start of the node. Extending this to a system with variables of arbitrary size is more difficult.

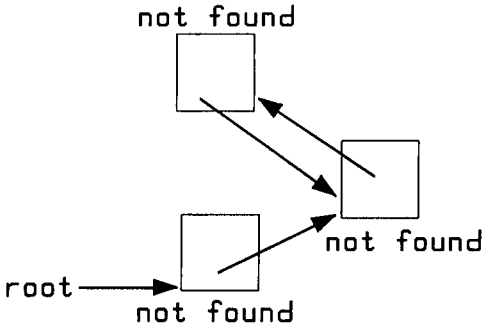
One possible method is to distinguish the start of a variable. This is possible if the variable's size is stored in the first word and it is tagged in a special way. By scanning the variables from the end towards the start, the end of the scan is given simply by detecting the start of the variable.

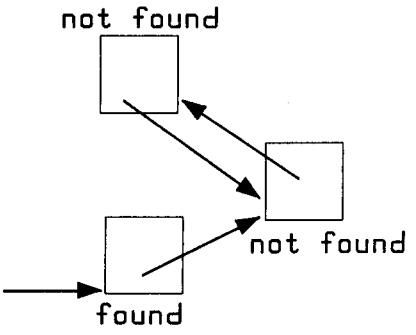
Schorr and Waite's algorithm is a technique for marking accessible variables. However, if the step which restores the reversed references is omitted then the algorithm can rearrange the memory so that the first word of each accessible variable contains the head of a list of all the references to the variable. The list is contained in the locations of the references themselves and ends with the original contents of the first word of the variable, probably its size field. This structure can then be used to update the references ready for compaction, as discussed in section 2.5.3.

2.3.3 Non Recursive Scanning

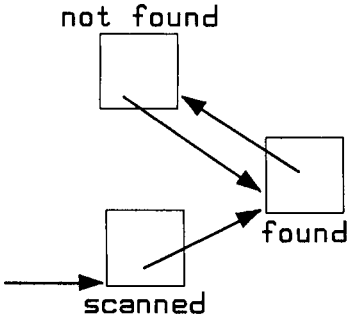
The stack required to control the scanning operation can be avoided altogether. This is achieved by making repeated scans of the memory to find accessible references to unmarked variables. This is the method adopted by [Dijkstra et.al.78]. Two bits per variable are required, one for marking whether a variable is accessible, the other for marking whether it has been scanned or not. In Dijkstra's algorithm the three states are described as **white**, **grey** and **black**. However, this thesis will use the more meaningful names **not found**, **found** and **scanned**.

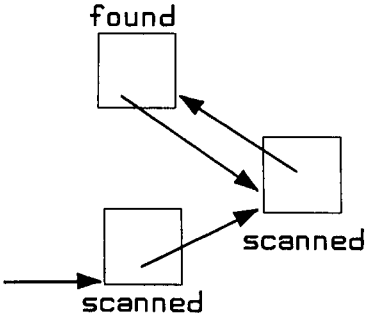
Figure 2.3a gives an example of a scan. Initially, all variables are marked as **not found**. Those which are directly accessible from the roots are then marked **found** and the scanning begins. A variable which is marked **found** is located and is searched for references. If any reference refers to a variable which is marked **not found**, the variable is marked **found**. Once a variable has been searched it is marked **scanned**.

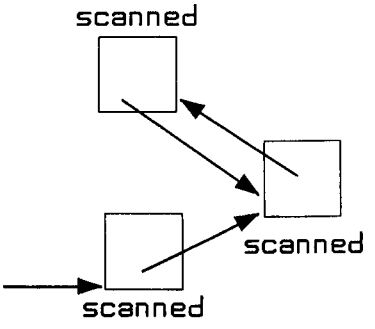




Initially all variables are marked as **not found**, then all directly accessible variables are marked **found**.







Next, **found** variables are located, in any order, and searched for references. If the reference refers to a **not found** variable, that variable is marked as **found**. After a variable has been searched, it is marked **scanned**.

Fig2.3a: Scanning is controlled by marks

This method is less efficient than reference reversal, because repeated visits to each variable in memory are required to find accessible unscanned variables, that is those marked **found**. The advantage, however is that the memory remains usable whilst scanning is in progress. For this reason it is suitable for incremental garbage collection.

Special requirements are placed on the computation to ensure that conditions for the correct operaton of the incremental garbage collector are maintained. All operations which move a reference must ensure that, during the scan, no **scanned** variables contain references to **not found** variables.

A variation of this algorithm, requiring just one bit per variable for marking, is presented by [Ben-Ari84]. However a severe performance penalty is incurred because variables marked as accessible are repeatedly scanned for references to variables marked as inaccessible.

To reduce synchronisation requirements when allocating new variables, Dijkstra's algorithm also scans the free list. Allocating a new variable is expressed simply as moving two references, which ensures the correct marking condition is maintained. However this technique is only possible if variables are of a fixed size.

The algorithm described by [Kung&Song77] is similar to Dijkstra's but variables are in one of four states, rather than three, and the free list is not scanned. The extra state introduced by Kung and Song is **new**. When variables are first allocated they are marked **new**. This allows variables to be created without synchronising with the garbage collector and without scanning the free list. This is most useful in systems which have variables of various sizes, where the free list is not simply a list of variables but a list of free store.

2.3.4 Two Memory Copying

Garbage collection of a memory can be achieved easily if a spare memory is available. All accessible variables are copied from the memory into the spare, where they are placed compactly. This leaves one area of free storage from which variables can easily be allocated. The roles of the two memories are then reversed, the first becomes the spare whilst the second becomes active.

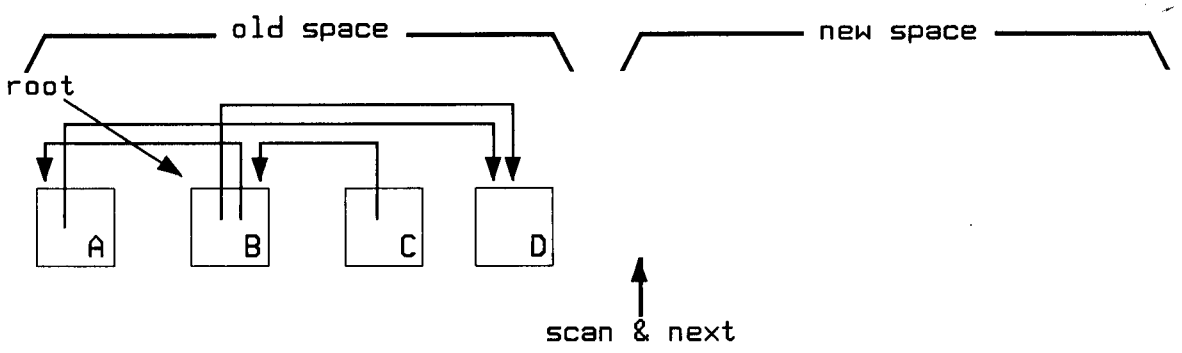
The copying process is recursive in nature and, since the memory is also compacted, it is necessary to update all references that are copied. The copying algorithm proposed by [Hansen69] is explicitly recursive and hence will require a stack to control the recursion.

Hansen's algorithm makes two intertwined passes across the memory. First all the references in a variable are found and the algorithm is applied recursively to the variables to which they refer. This gives the new location of those variables. The references are then updated and the updated variable is copied to its new location. Two bits are used to mark the variables. One indicates that the variable has been found and is being updated. When a variable has been moved, its new address is stored in the old copy and the second mark bit is set. A fixup table is used to cope with circularities.

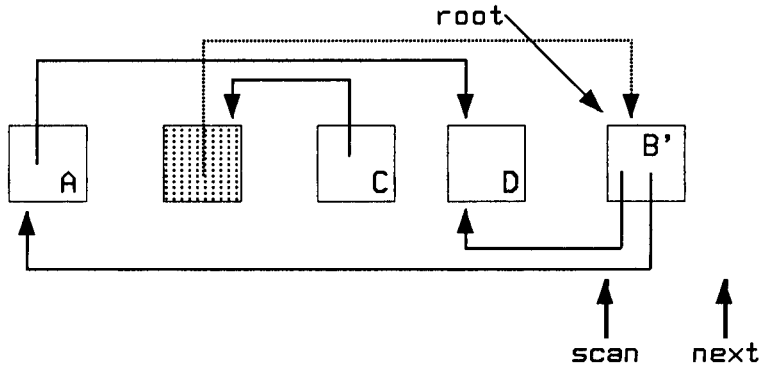
A similar algorithm for LISP is given by [Fenichel&Yochelson69]. Whilst their algorithm is recursive, they suggest that the [Schorr&Waite67] reference reversal method could be applied. This is where the space occupied by the references themselves is used to control the recursion, thus eliminating the need for a separate stack.

Another scheme is given by [Cheney70]. The references themselves are used to control the recursion, though in a much simpler fashion than Schorr and Waite's reference reversal. A version that does use Schorr and Waite's algorithm for reference reversal is given by [Reingold73]. Some improvements to this are suggested in [Clark76].

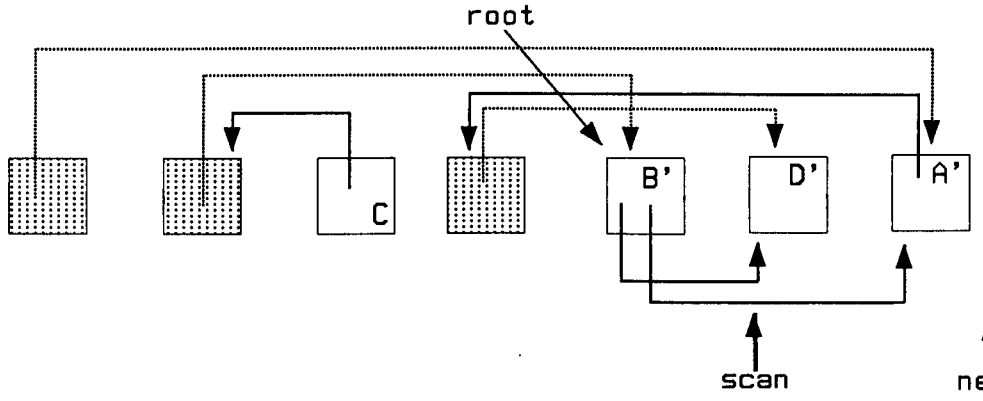
In Cheney's algorithm, two index variables, **next** and **scan,** which refer into the spare memory, are used. **next** indicates where the next variable to be copied is to be placed, **scan** indicates the progress of a single scan of the copied variables. Initially both are set to zero, then any variables known to be accessible are copied, with **next** being suitably incremented. The variables are copied without modification so any references refer back into the active memory. The sequence of diagrams shown in figure 2.3b illustrates Cheney's algorithm.



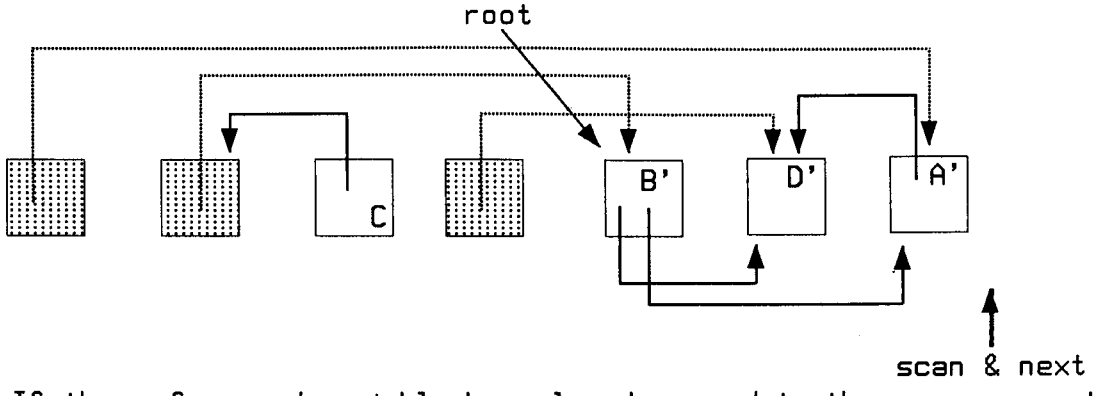
Initially both **scan** and **next** point to the start of the spare memory.



Variables which are directly accessible from the root are copied into the new space, leaving behind a forwarding address.



The new space is searched for references, using **scan** to control the search. If the referenced variable is still in the old space, it is copied into the new space and the reference is updated.



If the referenced variable has already moved to the new space, the reference is updated using the forwarding address. When **scan** reaches **next**, garbage collection has finished.

Figure 2.3b: Cheney's Algorithm

The scan now commences a search for references. When one is found the variable referred to is copied to **next**, if it has not already been copied. When a variable is copied a "forwarding address" is placed in the variable so that if further references to it are found they can readily be updated. The reference is then updated to refer to the new location of the variable and **scan** is advanced. The scan finishes when **scan** reaches **next**.

As well as needing an extra bit in each reference to indicate which memory is referred to, it must be possible to tell whether a variable has been moved or not. This can easily be achieved in systems with varying sizes of variables by using an extra bit stored with the variable's size field. Systems with fixed size variables, such as LISP, tend to have various flag bits stored in the variable, so it should not be too difficult to accomodate the 'moved' bit.

[Baker78] proposes using a two-memory copying garbage collector for real-time systems. Cheney's algorithm is used, since, unlike reference reversal techniques, the memory remains in a useable state whilst copying is in progress. Whenever a variable in the active memory is accessed a check is made to see whether the variable has moved to the spare memory. If so, the reference is updated and the access is made to the spare memory instead.

An interesting variation is proposed by [Ungar84]. This exploits the observation that many variables tend to be comparatively long lived. If a variable survives more than a certain number of garbage collection cycles, it is moved to another area which contains long lived variables. Inaccessible variables in this area are periodically reclaimed using a marking algorithm.

The copying garbage collector incorporated in the X2 object oriented virtual machine [Sandberg88] uses Cheney's algorithm rather than Baker's incremental algorithm because the pauses that occur are small enough to be acceptable. However in the LISP system described by [Moon84] an incremental version is used because the virtual memory is large and the pauses would be unacceptable.

Copying garbage collection algorithms are like scanning garbage collectors combined with compaction. The scan is controlled using a queue of variables which need to be scanned. This queue comprises the variables that have been copied into the new space but have yet to be scanned. The advantage of the copying collector is that no extra workspace is needed to implement this queue. The mark bit, which is usually explicit in the scanning algorithms, is replaced by an extra address bit which indicates which memory the variable is in.

2.3.5 Multiple Area Copying

Baker's algorithm is further developed by [Lieberman&Hewitt83]. They propose dividing the memory up into many areas, instead of just two, which are kept in order of creation. References within an area and from a younger to an older area are implemented normally. However, each area has an indirection table through which all references from older areas into the area must pass.

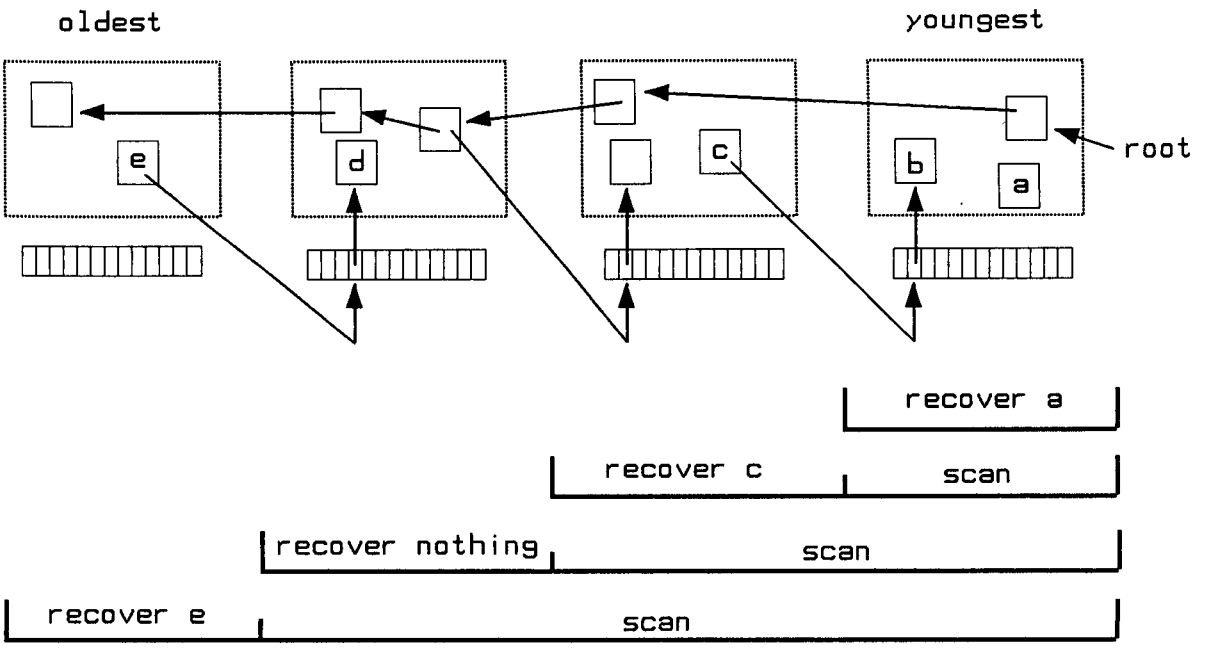


Fig2.3c: References to variables in younger areas pass through an indirection table. When an area is garbage collected, all younger areas must be scanned for references.

Garbage collection is started by copying the accessible variables in a area to a spare area, as in Baker's algorithm. However, to be sure that no references into the area remain, it is necessary to scan all younger areas looking for references to that area. Also, variables that are referenced through the indirection table are presumed to be accessible and are copied. Only when the scanning is completed can the storage occupied by the area be recovered and reused.

The example shown in figure 2.3c illustrates how references to variables in younger areas pass through an indirection table. If the youngest area is garbage collected, the variable 'a' will be recovered. If the second youngest area is garbage collected, variables in the youngest area will be scanned to find any references refering to variables in the second area. The variable 'c' will be recovered and a search of the youngest indirection table suffices to remove the entry for the reference to 'b'. Note that the variable 'b' will be recovered by a subsequent garbage collection of the youngest area.

Lieberman and Hewitt have observed that if a variable has been accessible for a long time, it is likely that it is relatively permanent and will continue to be accessible. Therefore the older areas tend to contain relatively permanent data and little garbage whilst the younger areas contain more garbage. It is therefore beneficial to garbage collect the younger areas more often. This is quicker than for the older ones since much less scanning is required. Therefore the rate of garbage collection of the areas can be varied according to their age to tune the performance of the garbage collector.

Cyclic structures can be created which cross area boundaries. If these become inaccessible then the simple algorithm fails to recover the storage. However, by copying an area and all those younger than it at the same time, it is possible to recover any inaccessible structures which cross area boundaries but are wholly contained in the copied areas. If a cyclic structure passes through the oldest and youngest areas then it is necessary to copy the entire active memory in order to recover its storage.

Lieberman and Hewitt assume that most references go from younger to older areas, and that cyclic structures are rare. This is true of LISP programs, for which the algorithm is intended, because the use of overwriting operations like rplaca is rare. However in general purpose computers overwriting is more common. For example, references from older to younger areas will occur when arrays of references are updated and items are added to queues. Therefore the assumption may not be valid in general.

2.3.6 Infrequent Garbage Collection

An interesting variation on the copying method of garbage collection is suggested by [White80]. This is to perform garbage collection very infrequently, say once a year, and in the meantime rely on a vast virtual memory system to supply new space. When space really does get low a large physical memory is used as a spare memory, into which the virtual memory is copied. White suggests this large memory could be hired from a garbage collection contractor, just for the duration of the garbage collection.

Whilst this approach seems attractive, especially with the advent of large density write-once laser discs, it is not clear that such a large virtual memory can be made sufficiently fast, in view of the sizes of page tables, or that it will be cost effective. In a distributed system, performing the annual garbage collection will be a sizeable task, involving the copying of incredible amounts of data, which does not appear very practical.

2.4 Multi-processor Garbage Collection

This section describes a wide variety of multi-processor garbage collectors. First to be considered are closely coupled systems, in which the processors share a common memory. In some of these systems processors are dedicated either to computation or to garbage collection, whereas in others the processors divide their efforts between the two tasks. Secondly, loosely coupled systems, those which have no shared memory, are discussed. Here each processor performs list processing on part of the distributed heap, and makes some contribution to the garbage collection of the whole heap.

2.4.1 Closely Coupled Multi-Processors

The use of two processors, one for list processing the other for garbage collection, was first suggested by [Steele75] as a way of avoiding the pause in list processing experienced when using most garbage collectors. The processors are very closely coupled, having shared access to the memory containing the heap.

In Steele's system, the garbage collection processor operates continually. It scans the memory marking accessible variables, using a stack to control recursion, and then returns any newly freed variables to a free list. The system is intended to run LISP in a virtual memory environment and so compaction is also performed, to reduce the size of the working set.

An analysis of a two-processor system, based on Steele's algorithm, is provided by [Wadler76]. Conditions are given which ensure that the free list is never exhausted thus allowing the list processor to run uninterrupted. Wadler concludes that incremental garbage collection requires twice as much processing power as those which require the computation to suspend. Similar analysis is provided by [Müller76].

A two-processor garbage collector was taken by [Dijkstra et.al.78] as an example in proving the correctness of a multiprocess program. Steele's original proposal used many semaphores to synchronize the two processors. Dijkstra attempts to limit the amount of synchronization required, thus keeping the list processor's overhead to a minimum. The algorithm is extended by [Lamport76] to allow more than one list processor and more than one garbage collector processor. A correctness proof for this is also given.

The results of a study of Steele's and Lamport's algorithms are presented in [Newman et.al.82] and [Newman et.al.83]. Some improvements to both are suggested which give significant speed increases.

An algorithm similar to Dijkstra's is used to recover garbage in the Cambridge CAP computer's filing system [Birrell&Needham78]. Most garbage is recovered by a reference counting technique. However, to recover inaccessible cyclic structures a separate process runs in the background which gradually scans the directory structure.

The Hydra multi-processor system [Wulf et.al.74] originally used a reference counting garbage collector. [Almes80] describes how Dijkstra's algorithm was adapted for use in Hydra's multi-processor environment to avoid the overheads of reference counting long lived variables.

Another application of the algorithm is found in the iMAX operating system of Intel's iAPX432 microprocessor [Pollack et.al.82]. Several, quite separate, heaps may be created, each garbage collected by a background process. In this microprocessor the scanning is performed by software, but the housekeeping operations, required to ensure correct incremental operation, are implemented in hardware.

Another two-processor garbage collector is described by [Kung&Song77]. This avoids the use of critical sections by relying on the mutual exclusion inherent in accessing the memory. A special queue is also used to hold references to all the variables that have yet to be scanned.

Another multiprocessor garbage collection system is described by [Halstead85]. This is for the Concert Lisp multiprocessor. Garbage collection is achieved using many Baker-style copying garbage collectors all working in one address space. The disadvantage to this algorithm is that all the garbage collectors must synchronise on swapping areas, which significantly reduces memory utilisation.

The chaining algorithm of [Newman&Woodward82] allows marking to be performed by several markers in parallel without the overhead of a stack and with minimal synchronisation overheads, and only one mark bit per node. However, the original form of this algorithm is very inefficient for cyclic structures and under certain circumstances does not terminate. An improved version is presented in [Newmanet.al.87] which employs the use of a small stack to overcome these difficulties. [Hudak&Keller82] propose a similar scheme which, like Dijkstra's algorithm, uses two mark bits per node. It is designed for an applicative system and many tasks can be spawned to mark the heap in parallel.

The performance of closely coupled multi-processor garbage collectors tends to be degraded by the overheads of synchronising and communicating between the garbage collector and computation, and by contention for the shared memory. An interesting two processor garbage collection system which is much less prone to these problems is described by [Ram&Patel85]. This exploits a paged virtual memory environment and contention only occurs for the pages on disc. Each processor has its own private primary memory, but access to the secondary memory is shared through a special purpose disc controller.

2.4.2 Loosely Coupled Multi-Processors

The garbage collector of the computer system described by [Bishop77] divides the heap into many areas. Each area is garbage collected using Baker's algorithm, though unlike Lieberman and Hewitt's algorithm it is performed quite independently on individual areas. Although the algorithm was described in terms of a single memory, it readily extends to a loosely coupled system [Ali84].

For each area two lists are maintained, one of all references that leave an area and one of all references that enter a area, as shown in figure 2.4a. Therefore each inter area reference is on two lists. Variables that are referenced from the incoming list are taken to be accessible. Along with those directly accessible from the roots of the heap, they form the starting point of a local garbage collection. If the local collection finds that a reference to a variable in another area is accessible, it marks the entry in the outgoing list. Once the local garbage collection has finished, any outgoing entry which has not been marked is removed from the outgoing list. In addition a message is sent to the area containing the corresponding incoming entry, informing it that the entry is no longer required.

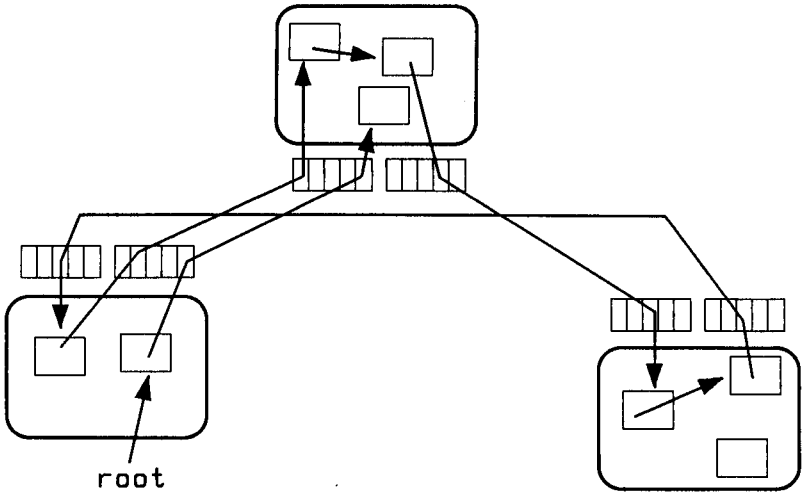


Fig2.4a: In Bishop's system inter-area references pass through an outgoing and an incoming indirection table. Each area is garbage collected independently, but inaccessible cyclic structures cannot be reclaimed directly.

Bishop's garbage collector also moves variables between areas, in an attempt to improve locality of reference. This is mainly intended to improve paging performance in a paged virtual memory, by reducing the program's working set, but also has the effect of moving inaccessible cyclic structures which span areas into just one area. Until this happens the garbage collector cannot recover the inaccessible store, because variables referenced from the incoming list are always assumed to be accessible.

This is not a satisfactory solution to the problem of recovering large inaccessible cyclic structures in a distributed system since users will usually want to keep control of the location of their variables. This is for efficiency reasons and because the semantics of remote and local variables may be be different. For example a computer may insist that instructions can only be taken from variables held locally. If the garbage collector moves such a variable in an attempt to localise cyclic structures, it will either cause the program to fail or thrash if the variable is moved back again by the computation.

Inter area references refer to variables through two indirections, the outgoing entry and the incoming entry. The entries are created when a reference is copied from a variable in one area to a variable in another area. This imposes a significant overhead on copying references between areas.

When a reference is copied between variables in the same area, no extra overhead is imposed. Also, if copies of an inter-area reference are made, then they will use the same link entry, again incurring no overhead. However if a reference to a variable in one area is copied between two other areas it is not possible to tell, without searching the lists, whether an entry already exists for that reference in the outgoing list. Either time must be spent searching the lists or a new entry must be allocated regardless of any duplication.

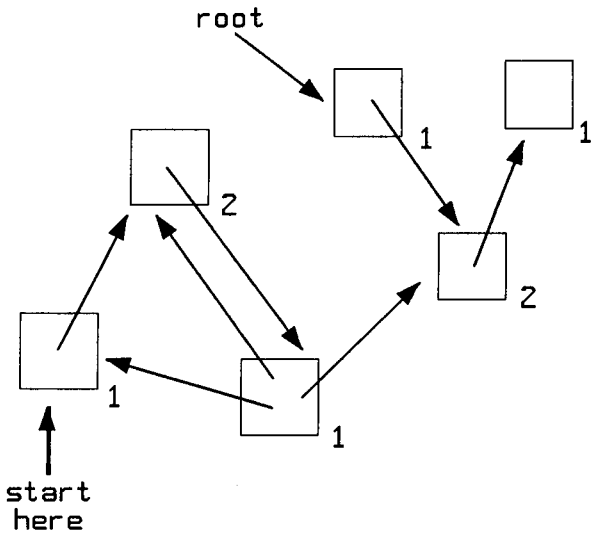
Maintaining the incoming and outgoing lists is therefore a significant overhead, either in time or space. In a distributed system, however, references are copied between computers relatively infrequently and the overhead is tolerable. The Flex computer system uses a hash table, instead of the simple linked lists, with fast microcoded searching to reduce the overhead. However, if the technique is used within a closely coupled system, the overheads may become significant in relation to the traffic in references.

To avoid the need for storing and maintaining list entries for references from rapidly changing areas, such as the temporary store of a process, to relatively more stable areas, such as the operating system, Bishop proposes the Cable. If an area A is Cabled to area B, then references in A can refer directly to variables in B. A consequence of this is that when area B is garbage collected, area A must be as well. However the garbage collection of area A can still occur independently of area B.

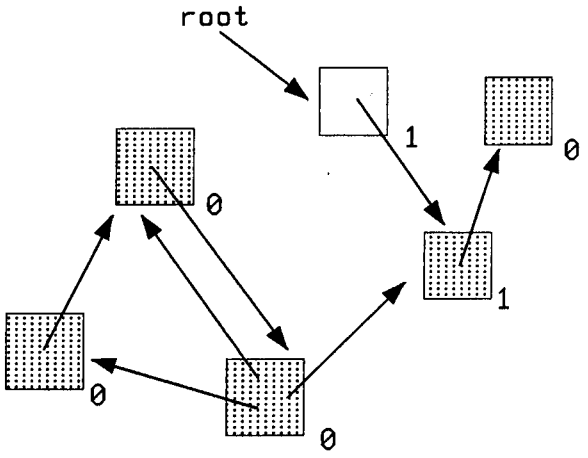
The problem with cables is that it is difficult to decide when to use them and that if they are used indiscriminately garbage collection can no longer be performed independently on each area.

The system proposed by [Vestal87] also divides the heap into areas, though reference counting is used to recover inaccessible acyclic structures. To handle cyclic structures, including those which span area boundaries, Vestal gives the following algorithm, which is illustrated by the example shown in figure 2.4b.

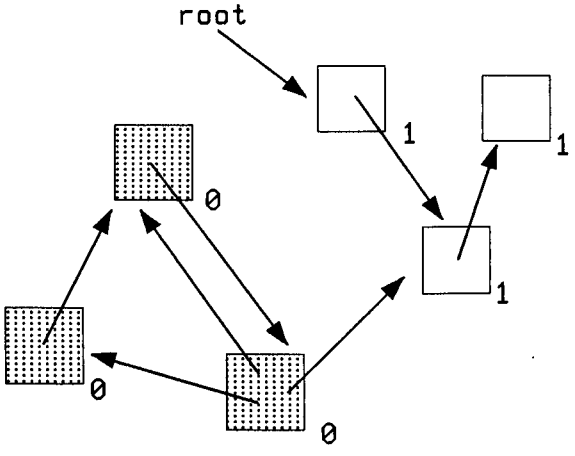
A set of potentially inaccessible variables is maintained. Initially this is empty, but a variable which is likely to be garbage is chosen, using suitable heuristics, and is added to the set. Each variable added to the set is searched for references. The variables referred to have their reference counts decremented and those which are not directly accessible are themselves added to the set. This phase of the algorithm terminates when all the variables have been scanned.



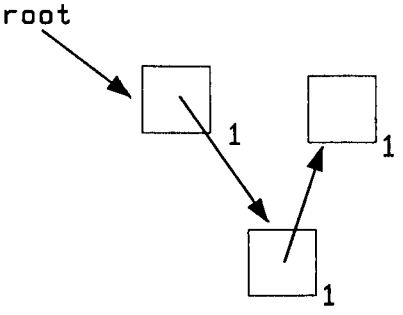
Initially, some variables are identified as potentially inaccessible.



The reference counts of these variables, and those accessible from them, are decremented.



Any whose reference count does not reach zero are restored.



Any whose count still remains zero can be recovered.

Fig2.4b: Vestal's algorithm can be used to determine whether some variables are part of inaccessible cyclic structures.

The second phase removes variables from the set if they do not have a zero reference count. Each variable removed is searched for references. The variables referred to have their reference counts incremented. This phase finishes either when the set is empty or contains only variables whose reference count is zero. These variables are now known to be inaccessible and are recovered.

The drawback to this approach is that it cannot guarantee to recover any garbage, so does not really satisfy the second essential criteria of effectiveness. There is also the problem of finding a suitable heuristic for choosing the first inaccessible variable. Vestal suggests this could be based on the times of creation and last use, but these are expensive parameters, in time and space, to maintain for each variable.

Another algorithm proposed by [Vestal87] uses scanning rather than reference counting. The heap is divided into areas, each of which is garbage collected independently. Variables referred to by other areas are assumed to be accessible. To recover these variables Vestal suggests a scheme similar to Bishop's. Variables which are only referenced from other areas are moved to one of those areas. If an inaccessible structure is moved so that it occupies just one area, it will then be recovered by the scanning algorithm. However, unlike Bishop's original algorithm, Vestal's version does not physically move variables. Instead the areas are formed from a logical grouping of variables.

The heap is garbage collected using a separate scanning collector for each area. Three states are used, **not found**, **found** and **scanned**, in a similar way to Dijkstra's incremental, non-recursive scanning algorithm. For each variable, a record is kept of its state in the garbage collection all the areas which may reference it, in addition to the area it resides in. An additional state is introduced which indicates that a variable is actually **unreachable** from an area. This is used to distinguish between a variable being identified as not referenced by an area and its state being set to **not wanted** ready for another garbage collection. Thus the recovery phase changes **scanned** variables to **not wanted** and **not wanted** variables to **unreachable**.

If a variable is **unreachable** from all areas which may reference it, it is inaccessible and can be recovered. However if it is **not wanted** by an area, its state may have just been reset ready for another scan, and so it cannot be recovered.

The scan of an area searches all variables which reside in the area and are marked **found** in the area's scan. It does not search variables in any other area. Therefore at the end of the scan, variables that reside in other areas may be marked **found** in the area's scan.

Once the scan of an area has finished, a variable which resides in the area and is marked **unreachable** or **not found** in the area's scan, but is marked other than **unreachable** in some other area's scan, is moved to that area. In doing so the variable is searched for references. Any variables it references which are marked **unreachable** or **not found** in the new area's scan are changed to **found**. Also if they are **not found** in the old area's scan they are changed to **unreachable**. These actions are necessary to ensure that the invariants of the garbage collections of the two areas involved are preserved. Once all such variables have been moved, any variables that remain in the area and are marked **unreachable** or **not found** in the area's scan are inaccessible and can be recovered.

Vestal's scanning algorithm, in common with Bishop's, cannot guarantee to recover inaccessible cyclic structures. This is because it is possible, though quite unlikely, that cyclic garbage will be moved round a ring of areas, each area attempting to localise the garbage by passing it on to the next. Vestal does suggest possible ways of reducing the probability of such an event occuring, but these effectively cause the garbage collections to synchronise, which nullifies the benefits of independent garbage collection of areas.

The store overheads of Vestal's algorithm are not inconsequential. Each variable requires an array of marks, one for each area, though if there are a large number of areas, the store requirement can be reduced by using a technique for handling sparse arrays, such as a hash table.

2.5 Compaction and Storage Allocation

2.5.1 Compaction

Once the inaccessible variables have been found, the storage they occupy can be returned to the free store where it can be used to allocate new variables. There are, broadly speaking, two types of organisation for the free store, where there is only one free block and where there is more than one.

Storage allocation from free store which consists of just one free block is easy. The variable is allocated from the start of the block and the block is made smaller. Garbage collection is required when the size of the free block is less than that of the variable requested. Returning inaccessible variables to such a free store is more difficult, since they are dispersed between the accessible variables. It is necessary to compact the accessible storage to one end of store, leaving one free block at the other end, as shown in figure 2.5a.

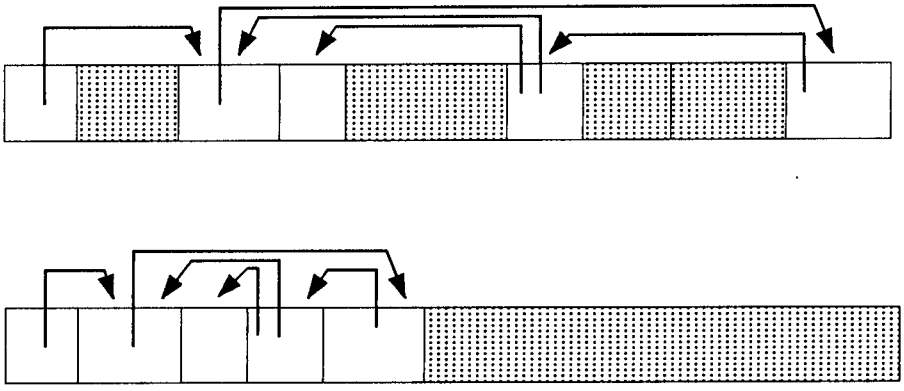


Fig2.5a: Compaction moves accessible variables together to create larger free areas. All references must be updated to refer to the variables' new positions.

A free store which consists of many free blocks can be constructed in several ways, the simplest being a linked list. Inaccessible variables are returned to the free store by adding them to one end of the list, though a study presented in [Harrold86] suggests it is better to add to the front of the list. To allocate a new variable, the list is searched for a free block which can accomodate it. It is then allocated from this block with any remaining free space staying on the free list. If no free block is large enough then garbage collection is necessary. However this may not result in a block which is large enough being found. The free store may actually contain enough memory, but be fragmented into many smaller pieces. In this case it is necessary to compact the memory to produce larger free blocks so that the allocation request can succeed.

A study by [Harrold86] shows the advantage of combining adjacent free blocks into one, and suggests an effective way this can be achieved using the tag bits which normally distinguish between scalar and reference data in a tagged capability computer.

Compaction is not required for systems which have a fixed variable size, such as LISP. However, it is sometimes used anyway to reduce fragmentation in virtual memory systems. This is because fragmentation effectively wastes part of the program's working set, causing more page faults to occur [Fenichel&Yochelson69].

Storage compaction involves moving some variables and updating all the references to those variables to reflect their new location. This may be done as a final pass of compacting garbage collection or may be a separate affair. Compaction is inherent in the copying style garbage collectors described in section 2.3.4.

2.5.2 Indirection Tables

The use of an indirection table to implement references greatly eases the problems of compaction. The table contains the addresses of all the variables in memory, whilst each reference contains the index, within the table, of the entry for the variable it points to. Whenever the variable referred to by a reference is to be accessed, the entry for that variable must be read from the indirection table to discover the variable's address.

If a variable is moved as the result of compaction then by altering the address in the indirection table, all references to the variable are simultaneously updated. It is not necessary to find all the references to the variable and update them individually. The disadvantages of the indirection table approach are that space must be found for the table and that going through the indirection table to reach a variable takes time. The latter problem, however, can be greatly reduced by using a simple cache memory or special mapping hardware [Tyner81].

If the maximum number of variables are allocated then the indirection table would be one third the size of memory. This is assuming a one word size field, one word indirection table entries and one data word per variable. Preallocating a table of this size is too wasteful of memory to be considered viable. Choosing a smaller size is a compromise between wasting memory and having enough entries available for peak demands.

Dynamically altering the space occupied by the table is possible, though it becomes necessary to be able to compact the table space as well as the memory space. However, this is much simpler since the entries are all the same size.

The use of indirection tables has not been given very much consideration is past literature. This is because previous work has centred on LISP systems, in which the variable size is always two. The use of an indirection table would therefore impose a serious overhead in time and space.

Indirection has been used in some capability computers, such as the Cambridge CAP computer and the Plessey PP250. Notably the Intel iAPX432 uses a two level indirection table to avoid the problem of preallocating enough table space.

2.5.3 Reference Updating

In a heap where references contain the address of the variable directly, rather than through an indirection table, it is necessary to find and update all references when a variable is moved. An algorithm for compacting such a heap was first given by [Haddon&Waite67]. While the accessible variables are moved, a table is constructed which gives the new location of each set of consecutive accessible variables. When all the variables have been moved a linear scan is made of the accessible storage. Any references are found and updated using the table.

Haddon and Waite show that no extra storage is required for the table, because it can always fit in the available free space. However, as compaction proceeds it becomes necessary to relocate the table. Improvements to this algorithm are proposed by [Fitch&Norman78] which speed up the accesses to the relocation table. [Berry&Sorkin78] show how the algorithm can be modified to give improved performance when the variables are allocated and discarded in a stack like fashion, as is usual for procedure activation records.

[Wegbreit72] gives an algorithm which updates all the references before moving any variables. The free variable located before a consecutive set of accessible variables is used to hold their new address. To update a reference it is necessary to find the first free variable preceding the variable referred to, since this gives the new address. This is accomplished by searching from the start of store until the free variable is found, though this search can be speeded up by constructing a directory.

The use of an extra address field in each reference is suggested by [Zave75]. This field is used to link together all references, sorted in order of the address of the variable they point to. The references are then updated in one pass by following this list. The store is then compacted.

The method of using reversed reference chains, which link all references to a variable together, to facilitate compaction was first suggested by [Fisher74]. Fisher's algorithm however, only works for systems in which the references all run in the same direction. [Morris78] gives a more general scheme. This uses two separate passes, one forwards and one backwards, in order to process both forward and backward references. A similar algorithm, which makes two forward passes, is given by [Jonkers79]. [Martin82] gives a faster version of Fisher's algorithm.

The algorithms of Haddon and Waite, Morris and Jonkers are compared in [Cohen&Nicolau83] using results obtained from a PDP10.

The compacting garbage collector proposed by [Thorelli76] also uses reversed references. However an extra word per variable is used to control the recursive scanning. The algorithm used in the Flex computer, [Fosteret.al.79], avoids the use of this extra word. This is done by adding all references, except the first, to the reversed list after the first reference. By ensuring that the first reference on the reversed list is the first that was found, it can be used to control the recursive scan, as originally proposed by Schorr and Waite. The references are updated in a separate pass, before a final pass compacts the variables. This makes the restrictions imposed by Fisher's algorithm unnecessary.

2.5.4 Storage Allocation

The allocation of new variables from a free store consisting of one free block is straightforward. However with multiple free blocks there are several possible allocation strategies. The free blocks are chained together on a linked list or in a tree structure so that they can be searched.

In the first fit strategy, the list is searched and the variable is placed in the first block found which is large enough to contain it. For the best fit strategy, the variable is placed in the smallest block which is large enough to contain the variable.

The cyclic placement strategy is similar to first fit, except that the search continues in a round robin fashion, rather than starting at the beginning each time a variable is allocated.

The different schemes are a compromise between the time taken to place an inaccessible variable in the free store, the time taken to allocate a new variable and the storage utilisation gained. Which scheme is best will depend on the pattern of storage usage in the computer. Many authors have modelled or simulated the various solutions: [Knuth73], [Campbell71], [Pflug84], [Page84], [Coffman et.al.85], [Baker et.al.85] and [Harrold86].

2.5.5 The buddy System

The buddy system was first proposed by [Knowlton65] as a fast method of allocating new variables of varying sizes, with minimum overheads for deallocation. A buddy system is initialised with its free store in one contiguous piece having a size which is some power of two. A separate free list is maintained for each possible size of fragment, which are restricted to powers of two.

To allocate a variable, its size is first rounded up to the nearest power of two. The extra space used by rounding request sizes is known as internal fragmentation [Randell69]. Next, an element is taken from the appropriate free list and is used to allocate the variable. However, if this list is empty an element of twice the size is taken and split into two. One half is used to allocate the variable and the other is put onto the appropriate free list. In fact, if the list of larger fragments is also empty, lists of still larger fragments are examined until a piece of free store is found. This is divided up until the allocation can be satisfied, with unused pieces going back onto the various free lists. This is illustrated in figure 2.5b.

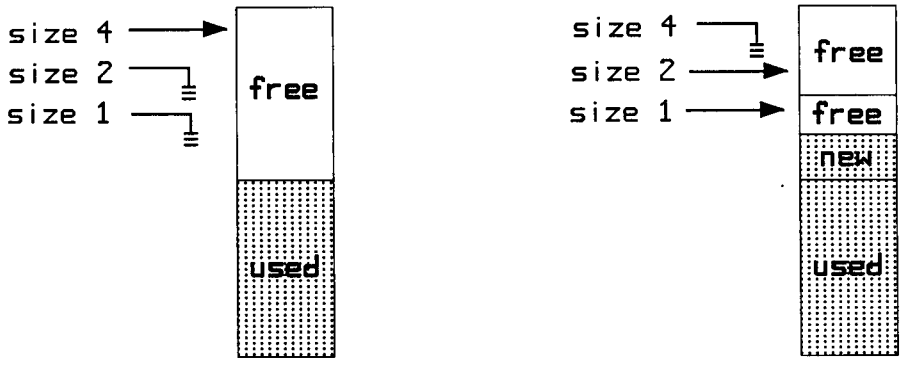


Figure2.5b: To allocate a block of size one, when only a block of size four is available, the free block is divided into two. One half is placed on the size two free list and the other is divided further. One half is placed on the size one free list and the other is used to allocate the new variable.

When a variable is deallocated, a check is made to see whether its "buddy", that is the other half of the store fragment from which it was created, is already free. This is done by searching the appropriate free list. If the buddy is found in the list, it is removed and joined to the newly released variable to form the original store fragment from which they were created. This process is repeated until the buddy is found to be still in use, in which case the free store fragment containing the newly released variable is added to the appropriate free list. Note that the address of a buddy is easily determined by inverting the address bit corresponding to the fragment's size, because the sizes are all a power of two. Figure 2.5c shows an example of releasing a variable.

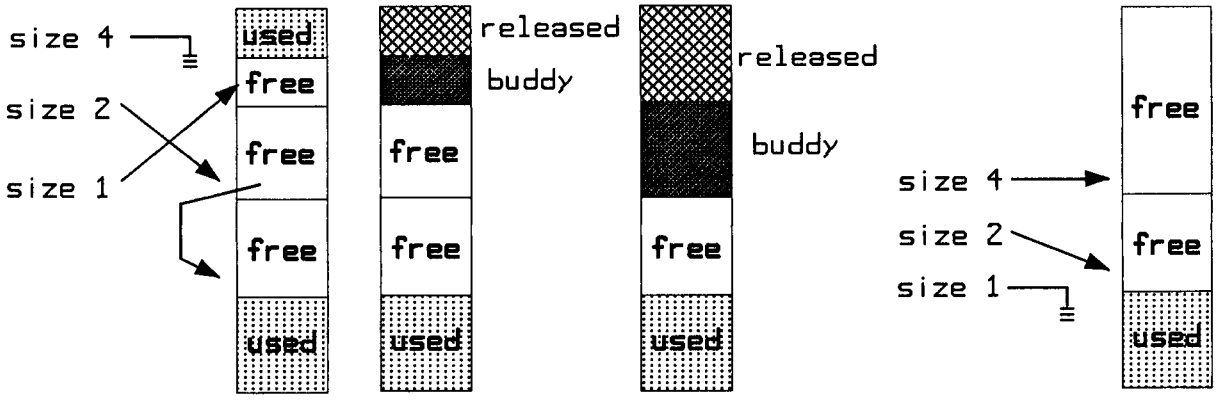


Figure2.5c: When the variable of size one is released, it is combined with its buddy which is already free. The resulting fragment's buddy is also free, and so these are combined.

Allocation and deallocation in the buddy system is relatively fast, but unfortunately it offers poor memory utilisation. This is because of both internal fragmentation and external fragmentation, which occurs when adjacent fragments are free but cannot be combined because they are not buddies.

Many variations have since been proposed, but anaylsis by [Peterson&Norman77] and [Purdom&Stigler70] shows them all to be fast, but with poor memory utilisation. [Randell69] reports evidence that rounding variable sizes up in an attempt to reduce fragmentation, as is necessary in the buddy system, actually reduces memory utilisation. [Page&Hagins86] and [Kaufman84] offer ways of tailoring the buddy system to particular patterns of use, in an attempt to improve performance. [Challab&Roberts87] show how detailed designs for various forms of the buddy system can be derived from more abstract algorithms.

2.6 Summary

The survey just presented has covered the two broad styles of garbage collection, namely reference counting and scanning, as well as briefly considering techniques for compaction and storage allocation.

Reference counting [Collins60] can be made reasonably efficient [Wise85], although [Ungar84] reports that reference counting in Berkeley Smalltalk consumes 15% of CPU time in managing reference counts, with an additional 5% taken in recursively freeing variables. It can be extended to work in a distributed environment [Watson&Watson87]. However, the technique suffers from the serious disadvantage that inaccessible cyclic structures cannot easily be recovered.

There are two kinds of scanning garbage collector. The marking collectors, typified by [Dijkstra et al.78], attach flags to each variable which mark its state in a scan that finds all accessible variables. This scanning can be made reasonably efficient [Harrold86]. The copying collectors, notably [Baker78], copy accessible variables into a free area, leaving behind any inaccessible variables. This technique inherently compacts the accessible storage, but this is inappropriate in a distributed system. Both varieties of scanning collector are able to recover inaccessible cyclic structures.

Some garbage collectors have been specifically designed for use in distributed systems. [Bishop77] applies a copying garbage collector independently to each computer in the system. Garbage collection of inter-computer references is handled using reference counts, and so inaccessible cyclic structures which are not within one computer cannot be recovered. Another algorithm, due to [Vestal87], offers a partial solution to this problem, but it cannot guarantee to recover the inaccessible cyclic structures.

	Counts	Marking	Copying	Recovers	Incremental	Parallel	Distributed
Collins60	•				•		
Deutsch&Bobrow76	•	•		•	•		
Hughes85	•			•	•		
Brownbridge84&85	•			•	•		
Wise85	•	•		•	•	•	
Watson&Watson87	•				•		•
McCarthy60		•		•	•		
Schorr&Waite67		•		•			
Dijkstra et.al.78		•		•	•	•	
Cheney70			•	•			
Baker78			•	•	•		
Lieberman&Hewitt83			•	•	•		
Steele75		•		•	•	•	
Bishop77			•		•		•
Vestal87$_1$	•				•		•
Vestal87$_2$		•			•		•

Fig2.6: Features of Garbage Collection Schemes

Figure 2.6 gives a table showing the main features of the more important garbage collection schemes given in the literature. All but two are incremental, in that garbage collector can suspend its activity and allow the computation to proceed. These two use reference reversal techniques and so the heap is in an inconsistent state while they are running.

Reference counting schemes cannot recover inaccessible cyclic structures, unless they use another technique as a fallback method, or only consider special cases. Marking and copying collectors are able to recover cycles, except in those algorithms designed for a distributed heap. Some marking collectors have been designed so that computation and garbage collection can proceed in parallel on different processors.

2.7 Conclusions

This survey of the literature has examined many garbage collection algorithms, covering the two broad styles of reference counting and scanning. Many algorithms have been tailored in some way to particular systems or programming styles, in particular LISP, which makes them inappropriate for use in general purpose systems. Other algorithms, such as the simple scanning and copying garbage collectors, are not suitable for use in distributed systems, because they do not scale up very well.

The most serious shortcoming of the reference counting algorithms is their inability to recover storage from inaccessible cyclic structures. It is not possible, in a general purpose system, to detect when these cyclic structures are created, so some technique must be employed to recover them. Applying reference counting to groups of variables is not very effective, because the cyclic structures can be quite large, and maintaining the groups places an unacceptable burden on the programmer, in much the same way as explicit deallocation in pools. These problems notwithstanding, maintaining the reference counts causes excessive memory accesses, which is likely to degrade system performance.

The scanning style of garbage collector can be made reasonably efficient, by using reference reversal to control the recursive scan and to update addresses for compaction. However this technique causes pauses in execution which are just tolerable in single user workstations, such as Flex [Foster et.al.82], but would be totally unacceptable when scaled up to a large distributed system.

Some scanning algorithms are incremental, in that the pauses are very short, but these take longer to complete a garbage collection. [Bishop77] and [Vestal87] have proposed using scanning garbage collectors as the basis for garbage collectors which would be suitable in distributed systems. These divide the heap into areas which are garbage collected independently, however they do not guarantee to recover inaccessible cyclic structures which span area boundaries.

The conclusion to be drawn from this survey is, therefore, that none of the surveyed algorithms is entirely suitable for the garbage collection of a general purpose, distributed heap store. The problem lies with the requirement for generality, given by C3 in section 2.1, in particular the ability to recover inaccessible cyclic structures.

However, with many of the algorithms described, a large effort has been expended on optimising them for particular applications. It would be most advantageous if the garbage collector of a distributed system allows individual computers to use the algorithm most suited to them to manage their part of the distributed heap. In this way the requirement for generality can be satisfied, yet at the same time advantage can be made of a special purpose garbage collector where it is appropriate.

Although no algorithm in the literature meets all the requirements, the proposals using independent areas appear the most appropriate. The problem is that the areas are garbage collected independently and inter-area references are handled by moving variables between areas. This thesis offers an alternative solution which uses a scanning garbage collector to manage inter-area references. Non recursive scanning is most appropriate for this, since the lengthy pauses produced by reference reversal are unacceptable and copying is inappropriate as compaction is not required.

Such a technique can hopefully be applied independently of the garbage collectors used by the computers in the distributed system. This allows the use of specialised algorithms where appropriate. In particular the algorithm itself may be used to garbage collect one of the component systems, if this were a smaller distributed system within the whole. This leads to the idea of a recursively structured heap, which is described in the next chapter.

3. Garbage Collection in a Recursively Structured Heap

This chapter presents the idea of a recursively structured heap store and develops an incremental garbage collector for it. The aim is achieve a better utilisation of memory, by recovering some inaccessible variables without garbage collecting the entire heap.

3.1 The Recursively Structured Heap

The heap is divided into areas which are garbage collected in parallel. The areas may themselves be divided further into more areas in a recursive fashion or may be garbage collected using any standard technique, either incremental or sequential. The use of areas does not restrict the locations in which references may be stored, so the user's view of the heap is the same regardless of its structure.

The structure of the heap may need to be carefully chosen to optimize garbage collection. For example, if the areas are chosen such that complex objects in the heap are contained within one area, inaccessible variables will be recovered more quickly than if they crossed area boundaries. However, the structure of the heap does not affect the garbage collection algorithm, only its efficiency and administration.

For the new algorithm, the heap, which may be large and distributed, is partitioned into disjoint logical areas. Each of these areas may in turn be divided into more areas, in a recursive fashion. Areas which are not sub-divided are called **leaf** areas, those which are are called **internal** areas. The entire heap is itself considered to be an area, called the **heap** area. An example of this structure is shown in Figure 3.1a. The **heap** area and each **internal** area are divided into one or more areas called **offspring** areas. Each **leaf** and **internal** area is an **offspring** of an area called its **parent** area.

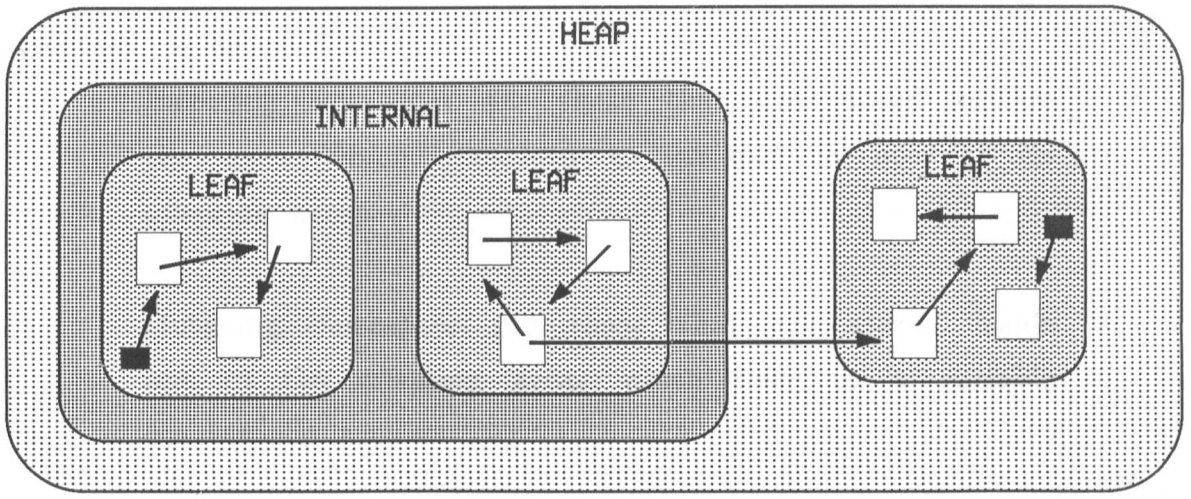


Fig3.1a Each variable, shown □, and each root, shown ■, belong to one leaf area. Areas are gathered together to form higher level areas and ultimately one area encompasses the entire heap.

The accessible structure in the heap is defined by the roots. These are a set of references to variables in the heap. They reside outside of the heap, for example in the registers of a processor. Each root location is considered, for garbage collection purposes, to be part of one of the system's leaf areas. However, not all leaf areas need contain any roots.

There are three kinds of references which concern an internal area, as illustrated in figure 3.1b:

Incoming references are those that are stored outside the area which refer to a variable in one of its offspring areas.

Outgoing references are those that are stored inside one of the offspring areas which refer to a variable outside the area.

Internal references are those which are stored in one offspring area but refer to a variable in another offspring area.

The case where a variable and a reference to it are stored in the same offspring area does not concern the area, because it is handled as an internal reference of some lower level.

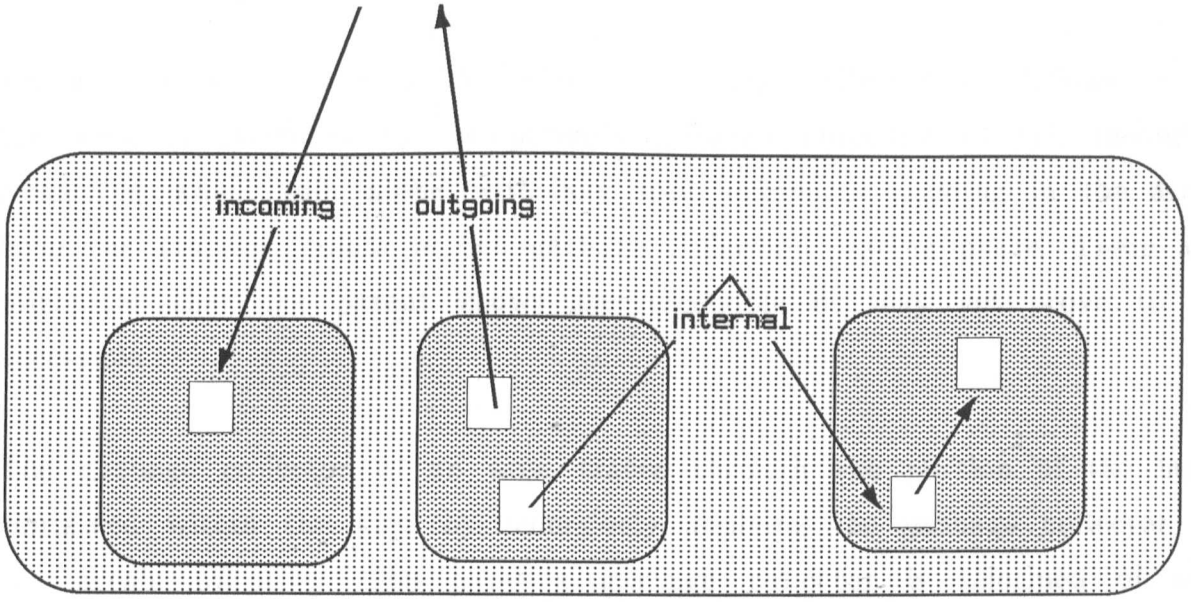


Fig3.1b Incoming References emmanate from outside an area, Outgoing References refer to a variable stored outside the area and Internal References are between different offspring areas.

To enable an area to be garbage collected, it is only necessary to maintain information about its incoming and internal references, though optimisations are possible which use information about outgoing references as well. In practice this information is likely to be stored in some form of indirection table. If more than one incoming or internal reference refer to the same variable, they may use the same indirection entry. However this is just an implementation detail, considered more fully in chapter four.

The parallel-recursive garbage collection algorithm will now be developed as a series of informal refinements, starting with a recursive sequential algorithm. This will be refined into a parallel version which is controlled recursively and finally into a completely parallel algorithm. For clarity, details of the interaction between the garbage collector and the programs using the heap are omitted. To operate incrementally, it is necessary for the programs to perform some housekeeping operations, which are described in section 3.5.

3.2 The Recursive Algorithm

An area which is not a leaf may be garbage collected as follows. First the area is informed by its parent's garbage collector of the incoming references which will, along with any root references residing in the area, form the starting points of the garbage collection tracing phase. These are references that the parent knows are accessible or may yet prove to be accessible.

The garbage collector of an area traces through the portion of the heap contained within the area by recursively applying the garbage collector to the area's offspring. The parent's garbage collector is informed of any outgoing references which are found to be accessible. Eventually, when the tracing has been completed, the state information for incoming or internal references that are no longer required can be discarded.

To control the tracing phase of the garbage collection, the incoming and internal references are marked with a flag. This takes the values **not found**, **found** and **scanned**. Initially references are marked as **not found**. When a reference is first found to be accessible by the tracing phase, its mark is changed to **found**. Once all the references in the variable referred to by a **found** reference have been followed, the reference is marked as **scanned**.

The recursive algorithm, for internal areas, is shown in figure 3.2a, using an algorithmic pseudo-language, described in Appendix B, along the lines of Pidgin Algol [Ahoet.al.74]. The garbage collection procedure takes three parameters. **wanted** is the set of references from which it must commence the tracing. **accessible** is a procedure used to notify the parent of any outgoing references that are found to be accessible from these. **keep** is the set of references which may still be needed by the parent. The area must keep these, and any variables and references accessible from them. However the parent must not be informed of any outgoing references which are found to be accessible only from the keep set.

```
garbage_collect = λ( wanted, keep : Set Ref,
                     accessible : Ref -> Void).
  refs = { r : Ref | internal( r ) or incoming( r ) }
  mark = λr:Ref . IF r.mark = not found THEN r.mark := found FI;
  trace = λ f : Ref -> Void .
    WHILE first time round OR ∃ r ∈ refs | r.mark = found
    DO FOREACH offspring a
       DO w = { r : refs | r refers to variable in a
                           AND r.mark = found };
          k = { r : refs | r refers to variable in a AND r ∉ w };
          FORALL r IN w DO r.mark := scanned OD;
          garbage collect( w, k, f )
       OD
    OD;
  { initialise }
  FORALL r IN refs DO r.mark := not_found OD;
  FORALL r IN wanted DO mark( r ) OD;
  { trace definitely wanted references }
  trace( λr : Ref . IF internal(r) THEN mark(r)
                                   ELSE accessible(r) FI );
  { initialise }
  FORALL p IN keep DO mark( p ) OD;
  { trace references which must be kept }
  trace( λr : Ref . IF internal( r ) THEN mark( r ) FI );
  { recover }
  FORALL r IN refs
  DO IF r.mark = not_found THEN recover space of( r ) FI OD
```

Fig3.2a: The Simple Recursive Algorithm

To illustrate how the garbage collector operates, consider the simple arrangement shown in figure 3.2b. This and subsequent figures are an example of how a garbage collection progresses. The garbage collector first marks all incoming and internal references as **not found**. Then those incoming references in the set wanted are marked as **found**. These form the starting point of the trace for references which are definitely accessible. In the example wanted is empty, because this is the outermost area.

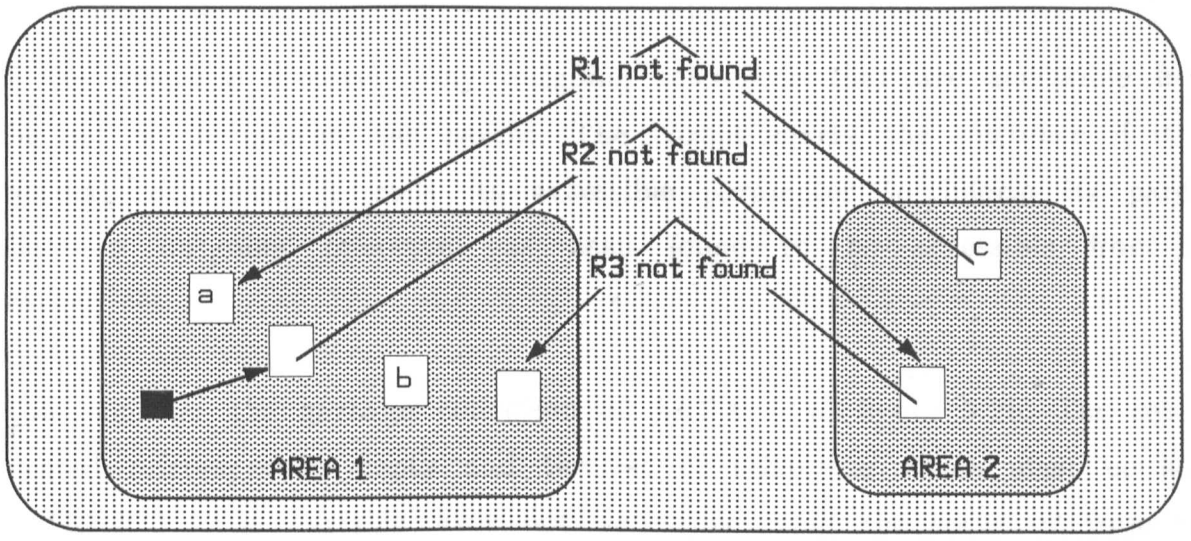


Fig3.2b The heap is divided into two areas. Initially all inter-area references are marked **not_found**. Variables a, b and c are inaccessible.

The trace is effected by garbage collecting each offspring in turn. This continues as long as any **found** references still remain, though each offspring must be garbage collected at least once in order to account for the root references they might contain.

When an offspring is garbage collected the set **w** is formed. This comprises of **found** references which refer to variables residing in the offspring. The set **k** is formed from the remaining references which refer to variables residing in that offspring. The references in **w** will form the starting points of the trace of definitely wanted references, and **k** is the set of references which the offspring must keep in case they are later prove to be accessible. The references in **w** are all changed to **scanned**, to indicate that the trace will have passed through them, and the offspring is then garbage collected.

If the offspring discovers any outgoing references that are accessible from the set **w**, it informs the parent. The parent marks the reference if it is internal, otherwise it calls its **accessible** parameter to inform its parent that an outgoing reference has been found to be accessible from its wanted set.

Now consider the garbage collection of AREA 1 in figure 3.2b. The set of wanted references is empty and the set of references which must be kept is {R1,R3}. The variable b is not accessible from the roots or from the wanted and keep sets and so is recovered. The variable a is accessible from the keep set. It is kept, although it is in fact inaccessible, because the garbage collector has to assume that it may yet prove to be accessible. The reference R2 is found to be reachable from the area's roots. Therefore the parent is notified and its mark is changed to **found**. Figure 3.2c shows the position after AREA 1 has been garbage collected.

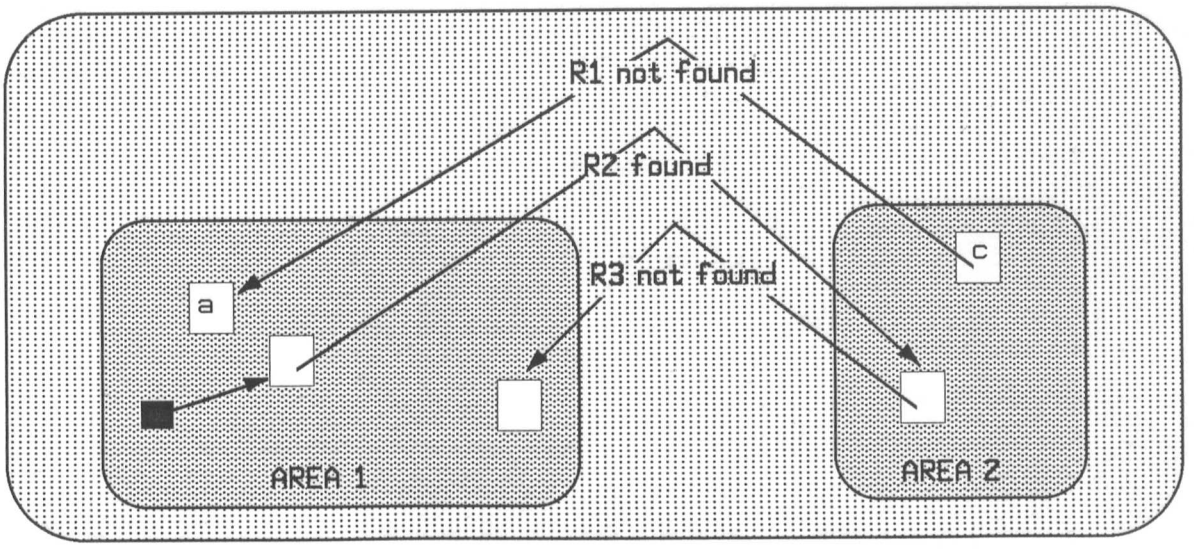


Fig3.2c After garbage collecting AREA 1, the variable b has been recovered and R2 is marked as found.

Next AREA2 is garbage collected. The set of wanted references is {R2} and R2 is changed from **found** to **scanned** to indicate that the trace will have passed through it. The set of references which are to be kept is empty. The variable c is not accessible and is recovered, but the reference R3 is found to be reachable from the wanted set. Therefore the parent is notified and its mark is changed to **found**. Note that, although no references now exist to the variable a, the area is unaware of this and it still maintains state information for the inter-area reference. An optimisation involving reference counts is discussed in section 4.6 which allows this to be recovered earlier. Figure 3.2d shows the resulting state.

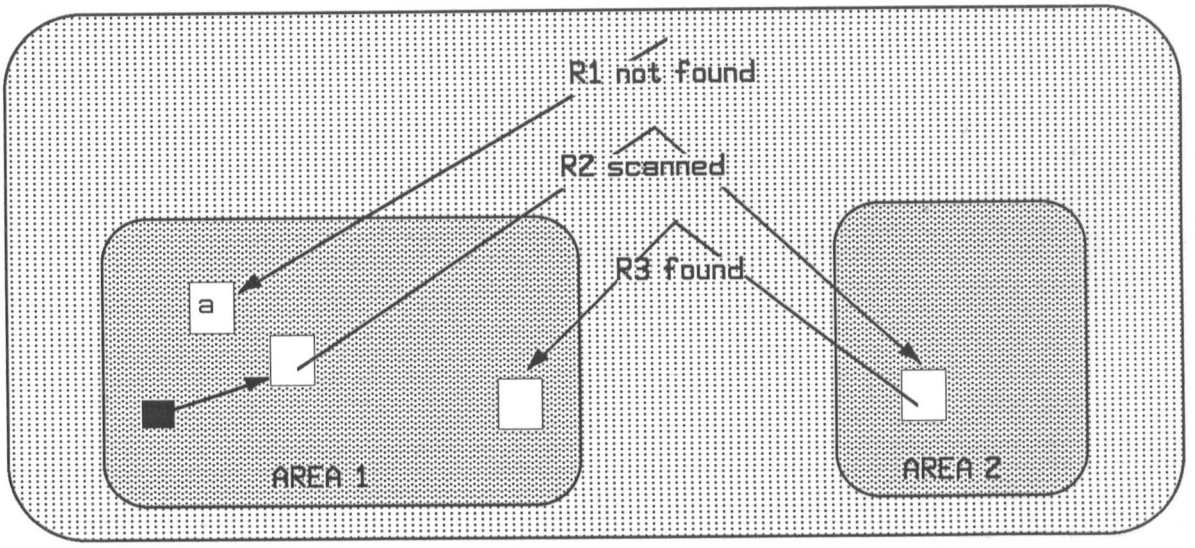


Fig3.2d After garbage collecting AREA 2, the variable c has been recovered, R2 is marked as scanned and R3 has been found.

Now AREA 1 is garbage collected again. The wanted set is {R3} and the keep set is {R1}. The mark for reference R3 is changed from **found** to **scanned**. The variable a is still kept, because the garbage collector still has to assume that it may yet prove to be accessible. Once this garbage collection has finished, no more **found** references exist so the first tracing phase has completed. Figure 3.2e shows the position after this has occured.

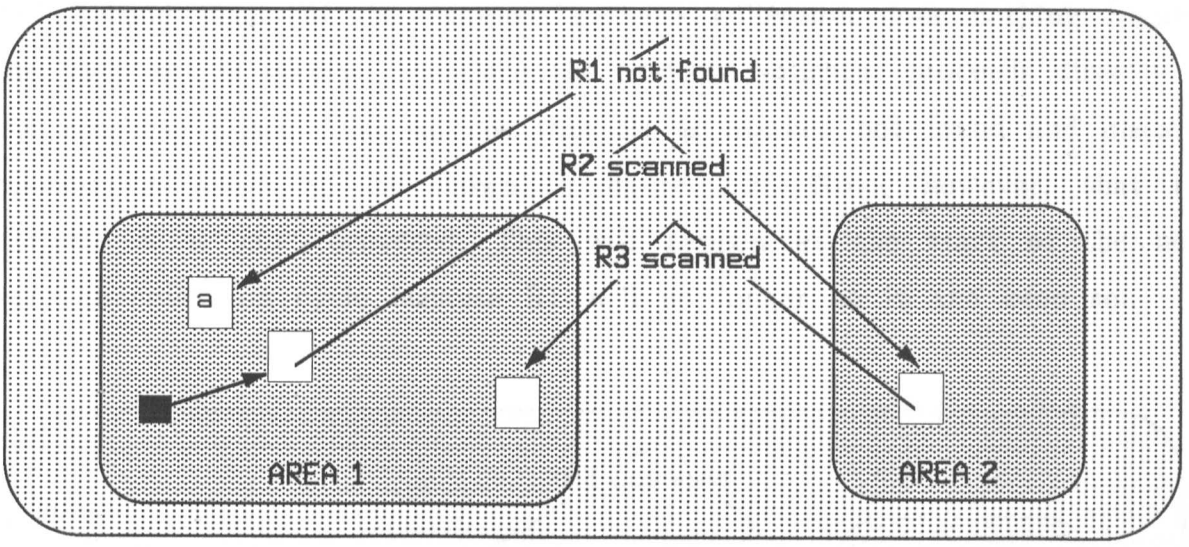


Fig3.2e After garbage collecting AREA 1 for the second time R3 has been scanned. The first tracing phase has been completed, because there are no references marked found.

Once no **found** references remain a second trace phase begins, but first those references in the set keep are marked. This trace is similar to the first, in that the offspring informs the area of which of its outgoing references are reachable. The area marks the reference if it is internal, but if it is outgoing its parent is not informed. This is because the reference is only accessible from the keep set, which the parent is not interested in.

The second trace phase finishes once no **found** references remain. Any references which are marked as **scanned** must be kept, but any marked **not found** are now known to be unreachable from either the wanted set or the keep set, and so they can be recovered. The final recovery phase completes the garbage collection of the area, but does not actually recover the inaccessible variables. This happens next time the offspring areas are garbage collected, because the recovered references which were keeping the variable alive no longer exist.

The second trace of the example garbage collection does nothing because the keep set is empty. Therefore the garbage collector proceeds directly to the recovery phase. This notices that R1 is marked **not found** and discards information about it, as shown in figure 3.2f. Note, however, that the variable a is not recovered. This will happen at the start of the next garbage collection cycle because R1 will no longer be in the keep set.

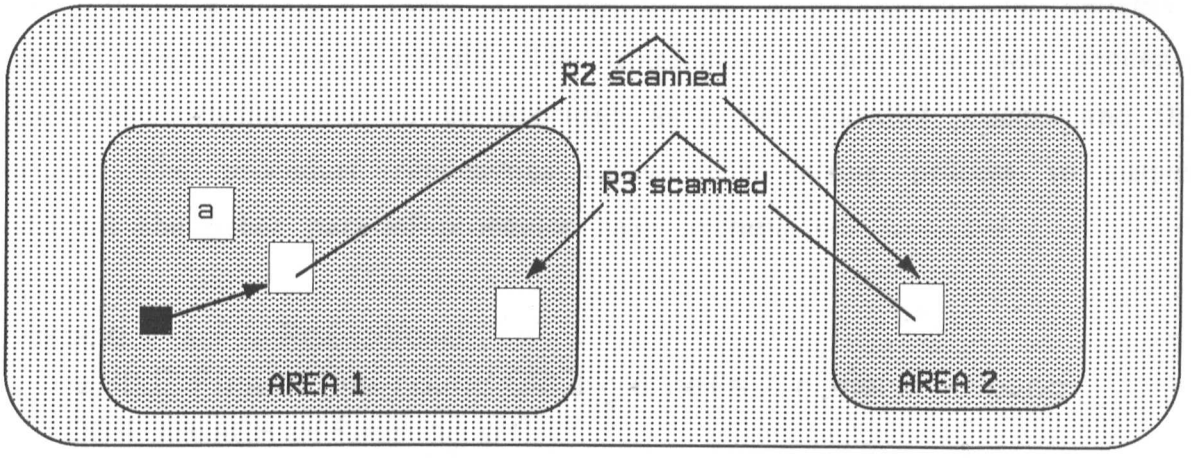


Fig3.2f Finally the information about reference R1 is discarded as it is found to be inaccessible. However the variable a is not recovered until the next time AREA 1 is garbage collected.

The example has shown that those inaccessible variables which have not been referenced from other areas, like b and c, are recovered quickly. However those which have been externally referenced, like a, take longer to be recovered. The assumption is that most garbage is local and the benefits of recovering this quickly outweighs the disadvantage of keeping global garbage for longer. This is investigated in chapter five.

Note that the abstract algorithm shown in figure 3.2a has a separate initialization phase which sets the marks of all references to **not found** and then marks those in the start set. In a practical implementation, this would be combined with the previous recovery phase.

3.3 Garbage Collection in Parallel

The simple recursive algorithm in figure 3.2a does not perform any of the garbage collection in parallel. This can easily be achieved by performing the garbage collection of the offspring areas in parallel with each other, as shown in figure 3.3a.

The procedure parallel takes a procedure as a parameter and delivers a similar procedure as a result. The difference is that the new procedure causes the original one to be executed as a new parallel process and then returns. The procedure rendezvous causes the calling process to suspend until all the other parallel processes have terminated.

```
trace = λ f : Ref->Void .
   WHILE first time round OR ∃ r ∈ refs | r.mark = found
   DO FOREACH offspring a
      DO w = { r : refs | r refers to variable in a
                          AND r.mark = found };
         k = { r : refs | r refers to variable in a AND r ∉ w };
         FORALL r IN w DO r.mark := scanned OD;
         parallel( garbage collect) ( w, k, f )
      OD;
      rendezvous
   OD
```

Fig3.3a: Performing the Tracing in Parallel

Greater parallelism can be achieved by continuously garbage collecting each offspring area, and not performing a rendezvous until the tracing is complete. This makes the trace's termination condition more difficult. Instead of just waiting until no **found** references exist, it is necessary to also ensure that each offspring area has finished locating all internal and outgoing references which are accessible from the **scanned** references. For this an extra boolean array is used to record whether the offspring is still tracing. This is shown in figure3.3b with the array called **ready**.

```
trace = λ f : Ref->Void .
  gc = λ i : Int .
    WHILE ∃ r ∈ refs | r.mark = found OR NOT ALL ready
    DO w = { r : refs | r refers to variable in offspring i
                        AND r.mark = found };
       k = { r : refs | r refers to variable in offspring i
                        AND r ∉ w };
       FORALL r IN  w
       DO ready[ i ] := FALSE; r.mark := scanned OD;
       garbage_collect( w, k, f );
       ready[ i ] := TRUE
    OD;
  FORALL r IN ready DO r := FALSE OD;
  FOR i TO number of offspring DO parallel( gc ) ( i ) OD;
  rendezvous
```

Fig3.3b: Continuous Parallel Tracing.

3.4 The Parallel Algorithm

The algorithm described so far is recursive. It shows how to garbage collect an area by garbage collecting its offspring areas. However the drawback of this algorithm is that if one offspring area takes a long time over tracing, other offspring areas are prevented from recovering any garbage. This is because, by the nature of the recursion, control of the garbage collection is invested in the higher level areas. What is needed is for areas to garbage collect as and when they find it necessary. The effects of these garbage collections must then be combined to produce the overall effect of garbage collecting the parent.

This can be achieved by constructing the address space recursively, but then launching parallel garbage collectors at the leaf areas. These would each garbage collect a leaf area, but they would run at their own rate. Additional processes are not required for internal areas since these are garbage collected by combining the effects of the garbage collection of their offspring.

The abstract form of this, the final parallel-recursive algorithm, is shown in figures 3.4a and 3.4b. The first gives an outline of a garbage collector for a leaf area. This takes two procedures as parameters. get_wanted is used to obtain, from the parent area, the set of references which are definitely wanted. These form the start of the trace for accessible store. In addition a procedure is supplied which is called for each outgoing reference found during the scan. get_keep is used to obtain the set of references which must be kept in case they prove to be wanted later. These form the start of a second trace, for storage that must be kept although it may not be accessible. The parent must not be informed of any outgoing references that are found during this scan, because these may later be shown to be inaccessible.

```
make_leaf_garbage_collector =
                λ( get_wanted : ( )->(Set Ref,Ref->Void),
                   get_keep   : ( )->Set Ref                    ) .
  garbage_collect = λ( ).
    FOREVER
    DO
      ( wanted, f ) = get_wanted( );
      trace from wanted, calling f for any
                                  outgoing references discovered;
      keep = get_keep( );
      trace from keep;
      recover inaccessible store
    OD;
  parallel( garbage_collect ) ( )
```

Fig3.4a: The Parallel-Recursive Garbage Collector for Leaf Areas.

The garbage collector for the leaf area is created and launched as a parallel process. The procedure which created it returns, allowing the other garbage collectors to be set up. The process first traces from the wanted set, calling found for any outgoing references that it locates. Next it traces from the keep set to determine all variables that must be kept in case they are still accessible. Finally any inaccessible storage is recovered and the garbage collection starts again. This continuous garbage collection proceeds at the appropriate rate for the amount of garbage generated in the leaf area, independently of the other areas.

The tracing phases of the leaf area's garbage collector may be implemented with either an incremental or a sequential algorithm. This allows systems to be constructed using components with differing styles of garbage collection. It is, however, necessary to make a small change to the garbage collection algorithms to enable them to scan in two phases and to notify the parent area of the discovery of accessible outgoing references during the first phase.

The garbage collector for an internal area is shown in figure 3.4b. A procedure is created for each of the offspring areas. This is responsible for coordinating the garbage collection of the offspring area with the garbage collection of the whole area. It is not, however, a separate process. Communication with the offspring garbage collector is controlled as a coroutine. That is two independent contexts are kept alive, very similar to processes except that only one runs at a time. Control is passed from one coroutine to another by calling special procedures. Data passed as the parameter of the call appears as the result of an earlier call in the other coroutine.

```
internal_garbage_collector =
                λ( get_wanted : ()->(Set Ref,Ref -> Void),
                   get_keep  : ()->Set Ref                    ).
  scan1done := array of bools all false, one for each offspring;
  scan2done := array of bools all false, one for each offspring;
  recover_done := array of bools, all false, one for each offspring;
  FOR r : Ref DO r.mark := not_found OD;
  accessible : Ptr -> Void;  {a procedure variable}

  make_collector = λ i: Int  .
    λ( trace : (Set Ref,Ref->Void) -> Void, keep  : ( Set Ref ) -> Void  ).
      mark = λ r : Ref . IF r.mark = not found THEN r.mark := found FI;
      found_ref = λ r:Ref.IF internal(p) THEN mark(r) ELSE accessible(r) FI;
      found_internal = λ r : Ref . IF internal to i( r ) THEN mark( r ) FI;
      refs = { r : Ref | r points into offspring i
                         AND internal to i( r ) OR incoming( r ) }
      FOREVER DO { trace store that's definitely wanted }
        scan2done[ i ] := FALSE;
        IF no other sibling has done so
        THEN  ( wanted, acc ) = get_wanted();    accessible := acc;
              FORALL r IN wanted DO mark( r ) OD FI;
        WHILE  ∃ r ∈ refs | r.mark = found OR NOT ALL scan1done
        DO  w = { r : refs | r.mark = found }
            FORALL r IN w
            DO scan1done[ i ] := FALSE; r.mark := scanned OD;
            trace( w, found_ref );
            keep( { r : refs | r ∉ w } );
            scan1done[ i ] := TRUE
        OD;
        { trace store that may be wanted }
        recover_done[ i ] := FALSE;
        IF no other sibling has done so
        THEN FORALL r IN get_keeps() DO mark( r ) OD FI;
        WHILE  ∃ r ∈ refs | r.mark = found OR NOT ALL scan2done
        DO  k = { r ∈ refs | r.mark = found }
            FORALL r IN k
            DO scan2done[ i ] := FALSE; r.mark := scanned OD;
            trace( k, found_internal );
            keep( { r ∈ refs | r ∉ k } );
            scan2done[ i ] := TRUE
        OD;
        scan1done[ i ] := FALSE;
        { recover }
        FORALL r IN refs
        DO IF r.mark = not_found THEN recover space of( r )
                                 ELSE r.mark := not_found  FI OD;
        recover_done[ i ] := TRUE;
        { wait for siblings to catch up }
        WHILE NOT ALL recover_done
        DO trace( refs, found_internal ); keep( {} ) OD
      OD;
  FOR each offspring i
  DO  coroutine( make_collector( i ),
                 either make_leaf_garbage_collector
                 or internal_garbage_collector as appropriate  )
  OD
```

Fig3.4b: The Parallel-Recursive Garbage Collector for Internal Areas.

The procedure make_collector constructs a collector procedure for offspring i. This collector procedure takes two procedures as parameters, similar to the leaf area garbage collector already described. In fact its structure is similar as well. It first traces the store that is definitely wanted, calling found for any outgoing references that it discovers. Then it traces store that must be kept in case it later proves to be accessible. Finally it recovers inaccessible store and starts a new cycle.

The internal area's garbage collector differs from the leaf area in that the store is being traced by the garbage collectors of the area's offspring. Also these traces occur in parallel. It is the coordination of these parallel threads that make the algorithm more complex. In particular, the threads must cooperate to ensure that the get_wanted and get_keep procedures are called only once for each cycle. Also, the threads must 'pause' after recovering inaccessible storage to ensure that all threads have completed the recovery.

3.5 Parallel Computations

The description just given in section 3.4 does not show how the algorithm operates incrementally. This is achieved with short critical sections, hence the pauses in execution of programs using the heap is very small.

If programs were allowed to manipulate the heap without synchronising with the garbage collector, it would be possible for the garbage collector to recover a variable even though there is still an accessible reference to it. This would occur when the computation copies a reference, which has not been found, into a variable which has already been scanned for references. If the computation then overwrites all copies of the reference which are stored in unscanned variables, the garbage collector will fail to find the reference. It will therefore assume the variable is garbage and recover the store it occupies. However a valid reference to it still remains, in the variable which had already been scanned, and so the heap is dangerously inconsistent.

To prevent this happening, the computation must cooperate with the garbage collector when references are copied. A sufficient condition is that no reference to a variable marked **not found** can be stored in a **found** variable, a **scanned** variable or a root. This can be met by checking the state of the variable whenever a reference is copied. If it is **not found**, the computation would mark it **found** before storing the reference.

This condition can be relaxed slightly if the computation can determine which **found** variable is currently being scanned by the garbage collector. A reference to a **not found** variable may be stored in a **found** variable, as long as it is not the one being searched. This is because the garbage collector will find the reference when it eventually searches the **found** variable.

When the computation proceeds in parallel with the garbage collection, the value given to a new variable's flag must be carefully considered. The obvious possibilities are **found** and **scanned**.

Setting the flag of new variables to **found** would mean that the garbage collector's termination condition may never be satisfied. This is because the computation may produce new **found** variables as fast as the garbage collector could search them.

The alternative, to mark new variables as **scanned**, creates problems during the recovery phase. Suppose a new variable is allocated and a reference to another variable is stored in it. If the recovery phase has already passed the new variable, it will leave it marked **scanned** but may reset the other variable's mark to **not found** ready for the next garbage collection cycle. This leaves a **scanned** variable containing a reference to a **not found** variable, which may cause the latter variable to be erroneously recovered.

The algorithm given by [Dijkstra et.al.78] avoids this problem by scanning the free list. This not only wastes time, but is inappropriate for systems with variables of varying sizes. However, it must be noted that the goal of the algorithm is to produce a system with the absolute minimum interaction between computation and garbage collector, which it achieves.

An alternative, suggested by [Kung&Song77], is to introduce a further state to the marks of variables, called **new**. Newly allocated variables are marked as **new**. The garbage collector treats this mark in a similar way to **found** during the first pass of the scanning phase. This accounts for any variables allocated during the previous recovery phase. Subsequently, **new** variables are treated in the same way as **scanned** variables. This is because they need not be searched, as they cannot contain any references to **not found** variables.

A third technique is possible if the computation and garbage collector are allowed to interact more closely. This is quite reasonable if they are implemented as coroutines in one processor. If the computation can determine which phase the garbage collector is in, it can mark new variables as **found** or **scanned** depending on whether the garbage collector is recovering or scanning.

Thus it is possible for the critical sections between computation and garbage collector to be very small. In particular, testing and updating a variable's flag must be performed as a critical section, as shown in figure 3.5.

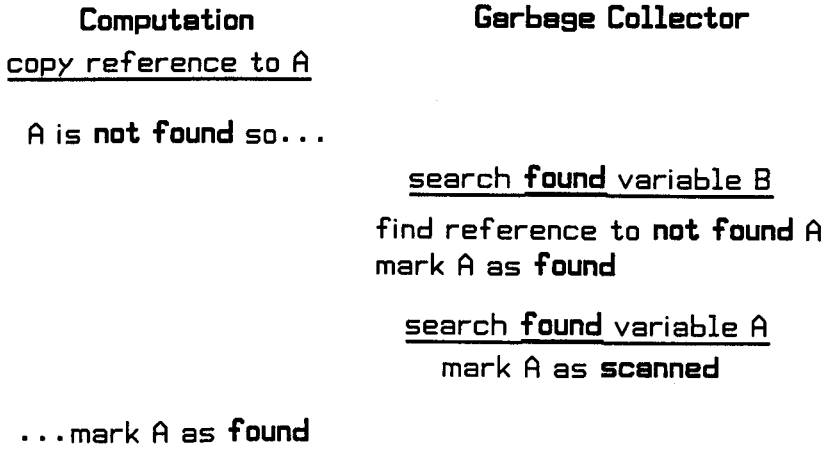


Fig3.5: Here is an example of the problems that can arise if the computation and the garbage collector do not use critical sections to manipulate a variable's flags.

3.6 An Example Garbage Collection

This section presents an example, to illustrate how the garbage collection of a recursively structured heap proceeds using the new algorithm. Consider the organisation shown in figure 3.6a. Here the heap area is split into two internal areas, labelled INTERNAL 1 and INTERNAL 2. INTERNAL 1 is in turn divided into two leaf areas, LEAF 1 and LEAF 2, and INTERNAL 2 is divided into LEAF 3 and LEAF 4. The system contains five variables, V1 to V5, of which all but V5 are accessible from a reference stored in the root, which is considered to be part of leaf area 1.

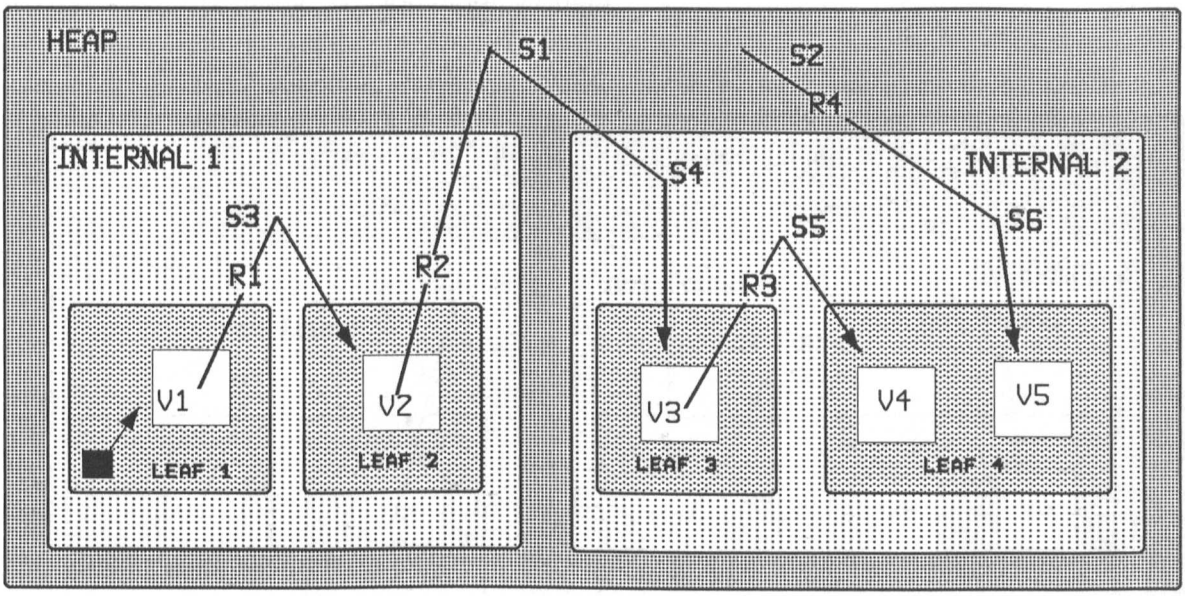
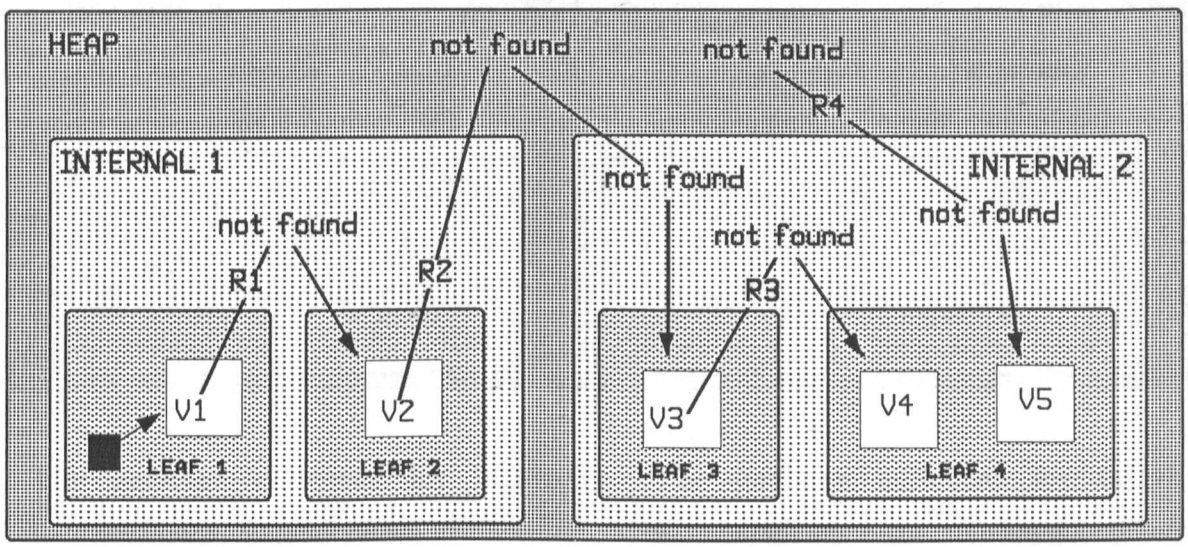


Fig3.6a: Example Areas and References.

There are four references, identified by the names R1 through R4. However each time these pass through a level of the heap's recursive structure, some state information is attached to them. This state information is represented in the diagram by S1 through S6. Each area maintains the state of all its incoming and internal references, but not for outgoing ones. Hence R2, which is outgoing for INTERNAL 1, internal for HEAP and incoming for INTERNAL 2, has two states, S1 in the HEAP area and S4 in INTERNAL 2.

The following diagrams and tables illustrate the progress of the example garbage collection. Initially all references are marked as not found. The heap area's garbage collector performs its first tracing phase by waiting for its two offspring to garbage collect.



INTERNAL 1
wanted = {}
keep = {}

The garbage collector for internal 1 performs its first tracing phase by waiting for its two offspring to garbage collect.

INTERNAL 2
wanted = {}
keep = {R2,R4}

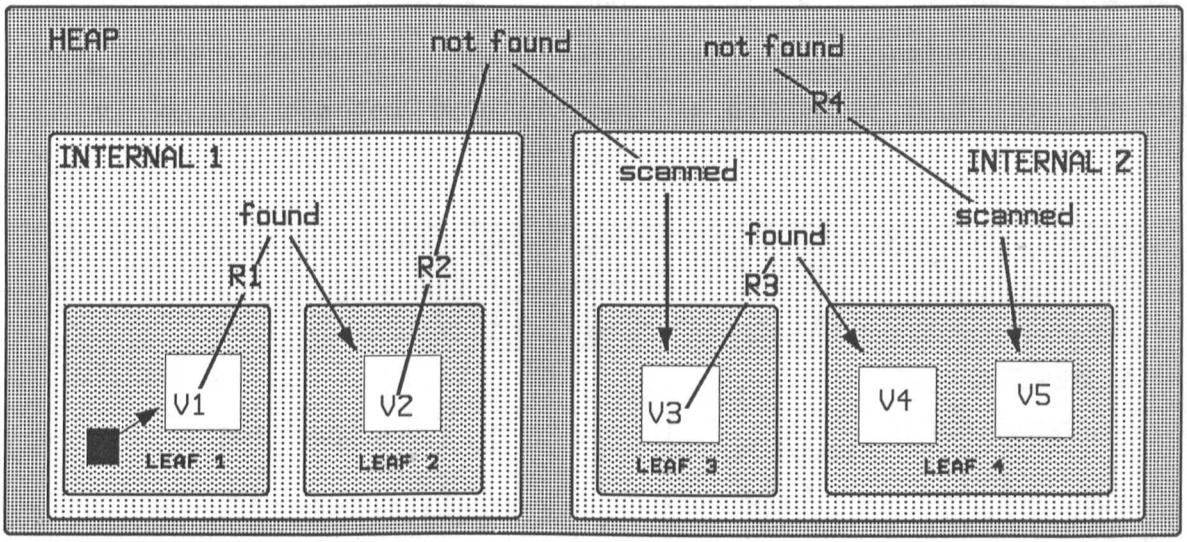
The garbage collector for internal 2 performs its first tracing phase by waiting for its two offspring to garbage collect.

LEAF 1
wanted = {}
keep = {}
accessible := {R1}

LEAF 2
wanted = {}
keep = {R1}
accessible := {}

LEAF 3
wanted = {R2}
keep = {}
accessible := {R3}

LEAF 4
wanted = {R4}
keep = {R3}
accessible := {}



The garbage collector for internal 1 performs its second tracing phase by waiting for its two offspring to garbage collect.

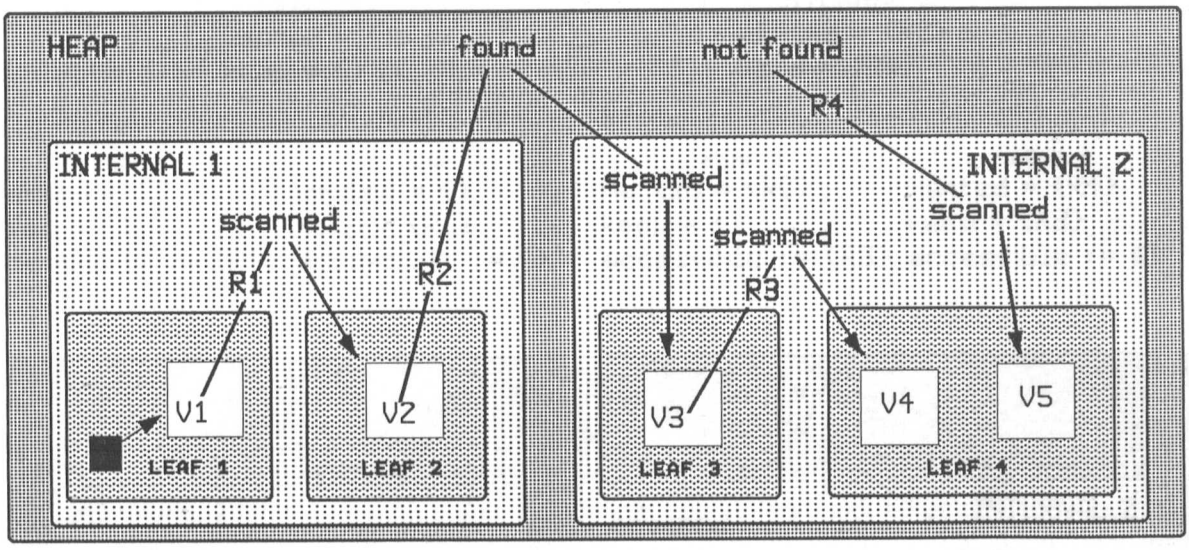
The garbage collector for internal 2 performs its second tracing phase by waiting for its two offspring to garbage collect.

LEAF 1
wanted = {}
keep = {}
accessible := {R1}

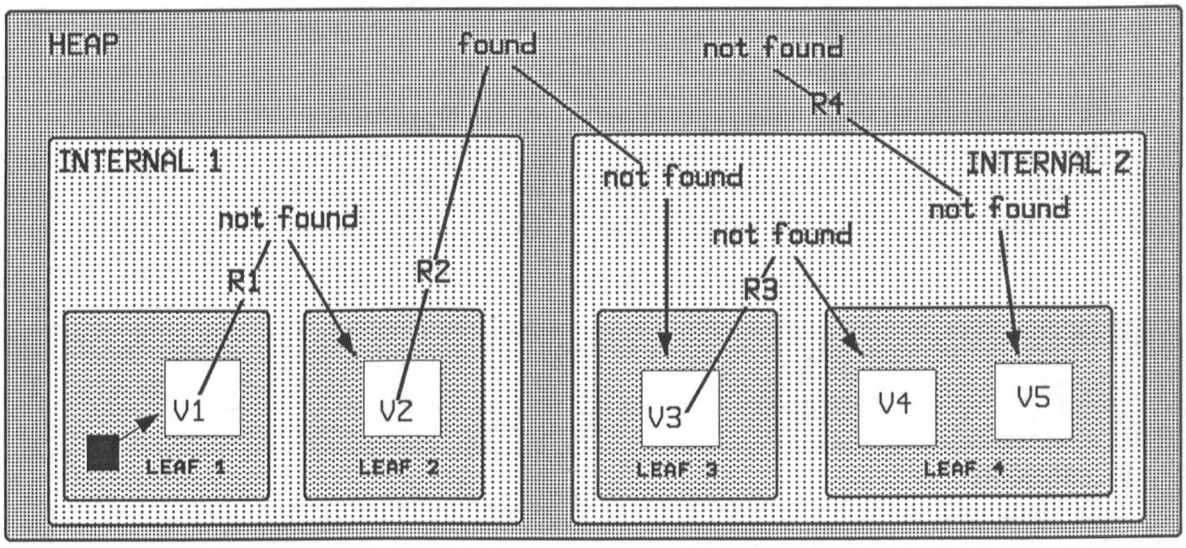
LEAF 2
wanted = {R1}
keep = {}
accessible := {R2}

LEAF 3
wanted = {}
keep = {R2}
accessible := {}

LEAF 4
wanted = {R3}
keep = {R4}
accessible := {}



The heap area's two offspring, internal 1 and internal 2, have now both finished garbage collection, because they have no references marked found. Their marks are changed from scanned to not found ready for the next garbage collection cycle.



The heap's first tracing phase has now completed. Next the heap garbage collector performs its second tracing phase, again by waiting for its two offspring to garbage collect.

INTERNAL 1
wanted = {}
keep = {}

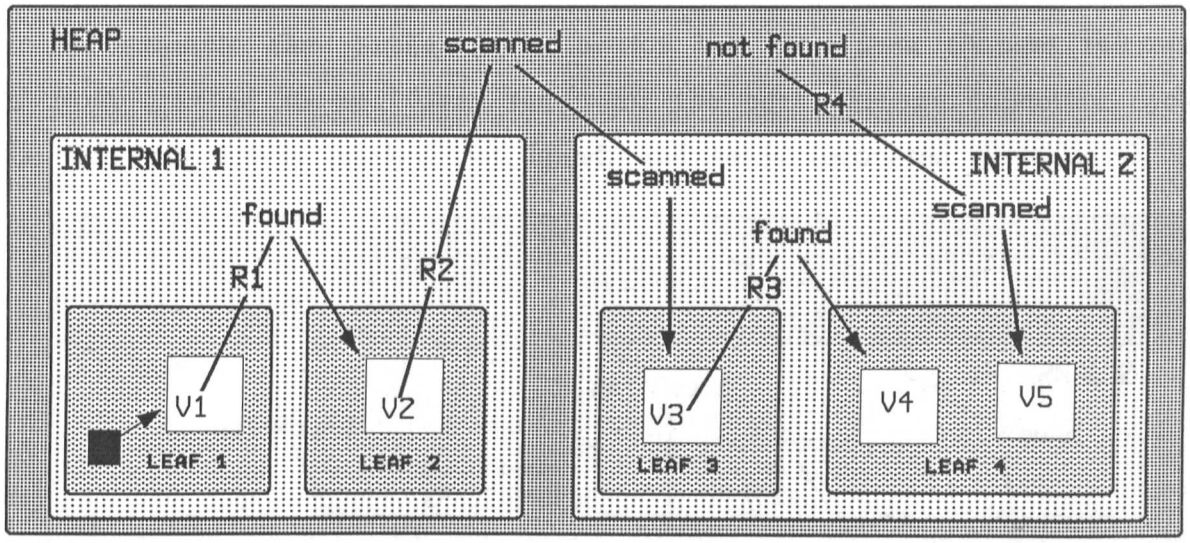
INTERNAL 2
wanted = {R2}
keep = {R4}

LEAF 1
wanted = {}
keep = {}
accessible := {R1}

LEAF 2
wanted = {}
keep = {R1}
accessible := {}

LEAF 3
wanted = {R2}
keep = {}
accessible := {R3}

LEAF 4
wanted = {R4}
keep = {R3}
accessible := {}

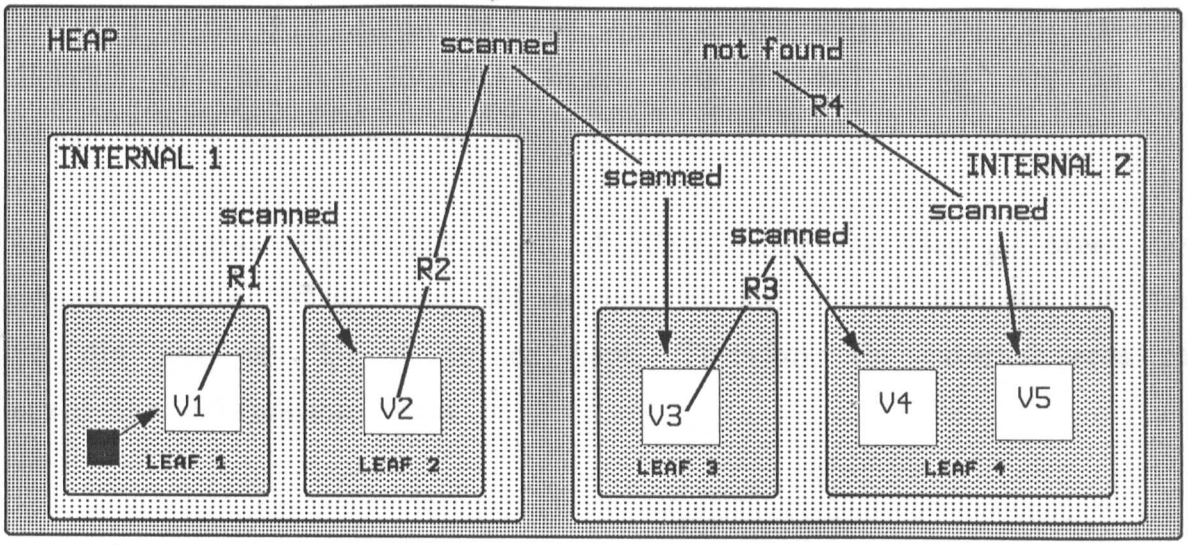


LEAF 1
wanted = {}
keep = {}
accessible := {R1}

LEAF 2
wanted = {R1}
keep = {}
accessible := {}

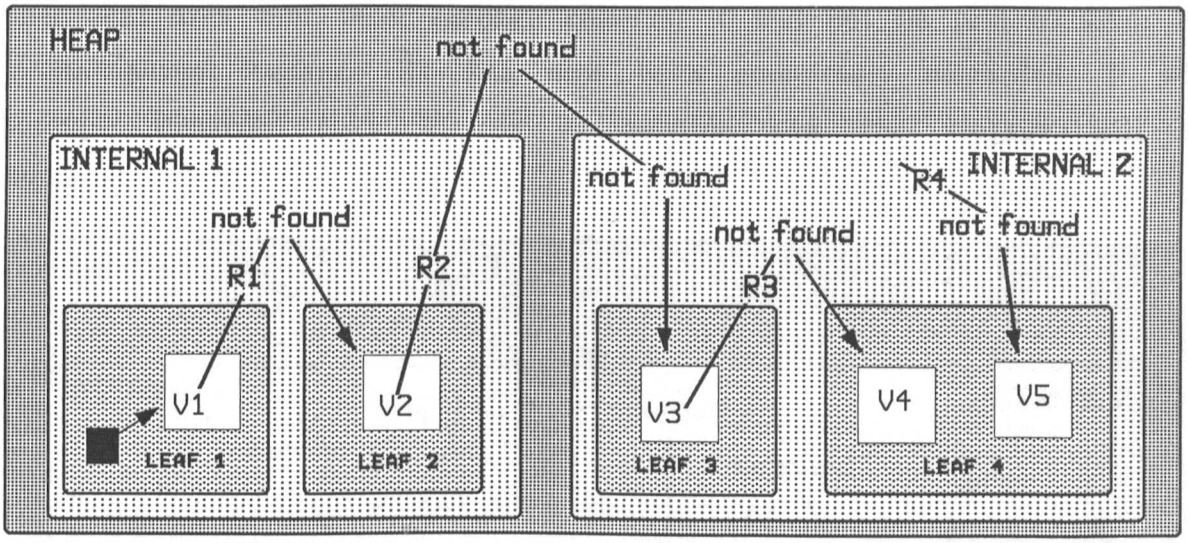
LEAF 3
wanted = {}
keep = {R2}
accessible := {R3}

LEAF 4
wanted = {R3}
keep = {R4}
accessible := {}



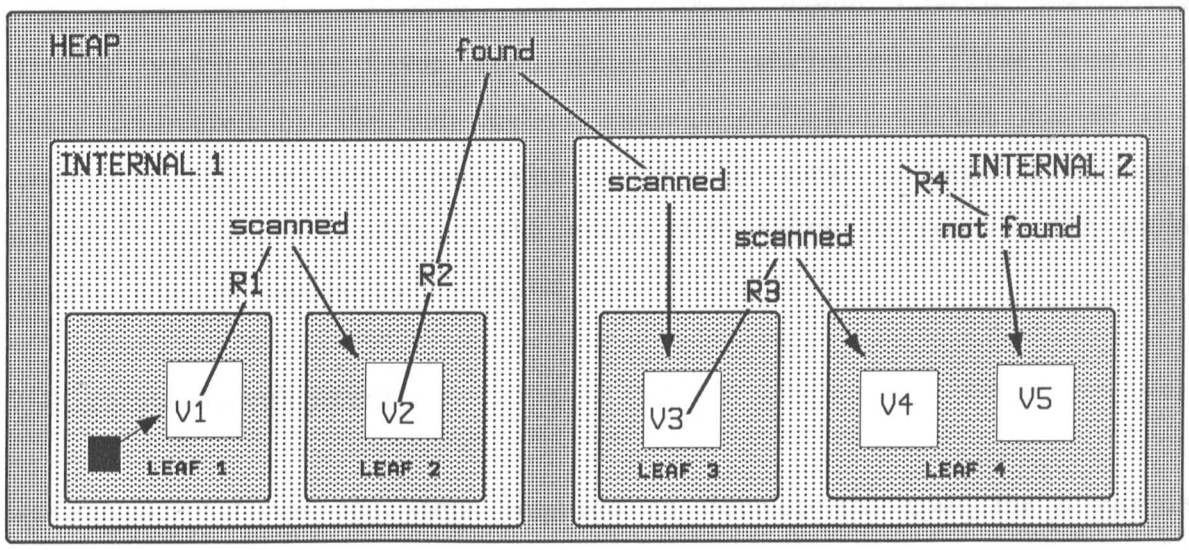
The two internal areas have now finished tracing because they have no references marked not found. Since the heap area has no references marked not found, it too has finished. The recovery phase of the heap area now recovers the space occupied by any inaccessible references. Note that any inaccessible variables will not be recovered until the offspring are garbage collected again.

In this example the garbage collector for the heap area changes reference R2 from scanned to not found ready for the next garbage collection cycle. The reference R4 is marked not found, and so is recovered.

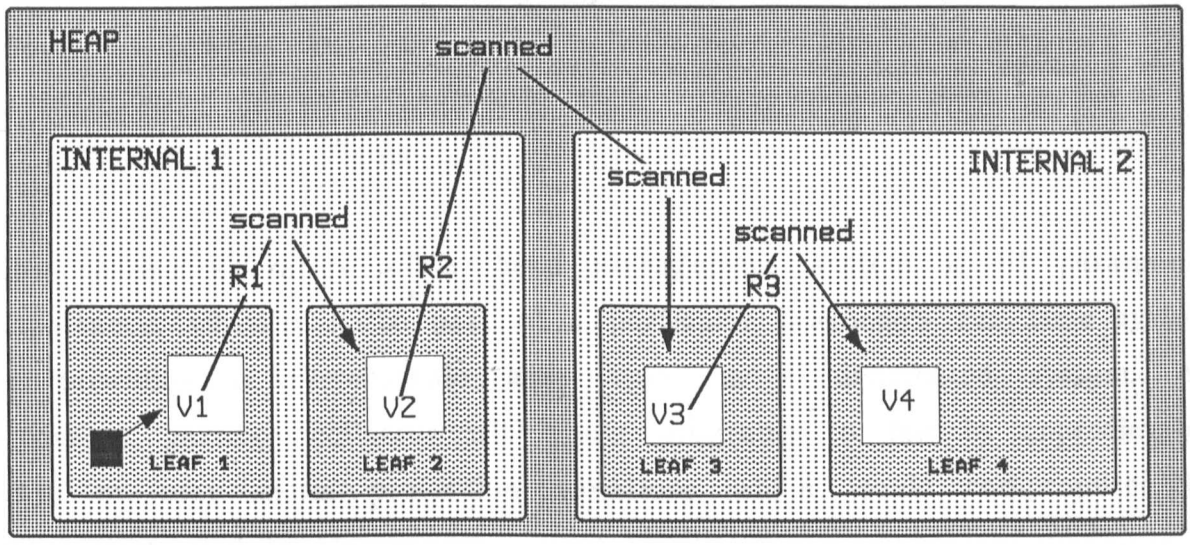


The heap area will again garbage collect by waiting for the garbage collection of the two internal areas. The internal areas will in turn garbage collect by waiting for the leaf areas to garbage collect.

The next diagram shows the state of affairs after the internal areas have garbage collected. This time R4 is marked not found, because the heap area did not ask for it to be kept. Therefore the internal area can discard the information about it.



When leaf area four next garbage collects, it will not be told to keep V5 by its parent because R4 has now been completely discarded. It will therefore discover that V5 is inaccessible and the storage will be recovered.



Thus the variable V5 which was inaccessible at the start has eventually been recovered. Each leaf area has been garbage collected many times and this contributed to the scanning of the internal areas, which in turn effected the scanning of the whole heap.

In a more realistic example, the leaf areas would contain inaccessible variables which have never been externally referenced. These would be quickly recovered by the independent garbage collection of the leaves. It is hoped that the relative speed with which this local garbage is recovered outweighs the disadvantage of the extra time taken to recover the global garbage. This is analysed in chapter five.

3.7 Rigorous Development

This section outlines the steps required to develop the code for a garbage collector by refining its specification. Part of the garbage collector is formally specified and a small fragment of this is refined into code to serve as an example. Even this modest exercise is quite lengthy, which shows that full specification and rigourous development to the implementation language, possibly microprogram, will require considerable effort.

3.7.1 The Need for Rigour

The correct operation of a heap management system is of paramount importance, whether it be part of a particular language's run time system or providing a system-wide heap store in a distributed system. An erroneous garbage collector will cause programs to fail in unpredictable ways or may cause protection violations which would compromise the information held in the system.

In the first instance it may be difficult to ascertain that it is the garbage collector which is at fault, rather than the program using the heap. However, even when the culprit has been identified, mistakes are virtually impossible to rectify using traditional debugging techniques. This is because of the volume of information involved and the difficultly in reproducing apparently random effects. These factors suggest that development of a garbage collector would benefit from the rigorous approach to program development [Jones80], [Gries81] and [Backhouse86].

Each proponent of this approach offers slightly different methods, but essentially the idea is that verifying that the implementation meets its specification should proceed hand in hand with the construction of the program. The method is described as rigourous because the proofs are precise and accurate, just like standard mathematical proofs. A formal proof, in the sense that every step is given in meticulous detail, could be produced by machine [Craigen85], but the cost of producing it may outweigh the increase in confidence about the program's correctness.

One area which is still in the research stage is the treatment of parallelism [Milner80] and [Hoare85]. However, it is likely that, in many practical implementations, a single processor will switch between garbage collection and normal computation. This switch will only be allowed to occur at certain well defined points, which means the interaction can be described in terms of coroutines. Therefore, it is not necessary to employ the full power of parallel processes to specify or implement the algorithm.

The informal correctness proofs given by [Dijkstra et.al.78], [Kung&Song77] and [Lamport76] are for similar incremental garbage collectors. However these programs have not been developed along with their proof, rather the program was written and then proved correct. While this is quite valid, it is more difficult to achieve.

The code for the garbage collector must, therefore, be developed along with its proof of correctness. The proof need not be completely formal, but can contain elements of common sense, as is usual with mathematics.

3.7.2 The Specification

To illustrate how the rigorous development would proceed, a garbage collector for a leaf area will be specified and refined. The specification will be written in the language Z [Hayes87]. This is based on elementary set theory and provides a means of structuring specifications in an incremental style. A glossary of the symbols used is given in Appendix A.

The specification starts by introducing two sets which represent references and variables.

$$[Ref, Var]$$

Some identifiers are now declared which, with suitable predicates, form a description of the heap.

```
incoming, internal: Ref ⇻ Var
outgoing : ℙ Ref
contains: Var ↔ Ref

roots: ℙ Ref
vars : ℙ Var
```

```
outgoing ∩ dom( incoming ) = {}
outgoing ∩ dom( internal ) = {}

ran( incoming ) ∪ ran( internal ) ∪ dom( contains ) ⊆ vars
roots ∪ ran( contains ) ⊆ dom( internal ) ∪ outgoing
```

Two partial injective functions, **incoming** and **internal**, map references to variables. Note that the same reference may be in the domains of both functions. Some references are **outgoing**, that is they refer to variables in other areas, and not to variables in this area. The **roots** may contain references to variables in other areas or to those in this area, given by **vars**.

The schema COLLECT defines which variables are to be recovered, given a set of references which are known to be wanted and a set which must be kept in case they later prove to be accessible. It also defines those references which are found to be accessible from the roots or wanted references.

```
COLLECT
wanted, keep, accessible' : ℙ Ref
recover' : ℙ Var
---
wanted ∪ keep ⊆ dom( incoming )

  accessible' = outgoing ∩ w
  recover' = vars \ (incoming ∪ internal)(| w ∪ k |)
where
  w ≙ ( internal ; contains )* (| roots ∪ wanted |)
  k ≙ ( internal ; contains )* (| keep |)
```

Those references which are wanted and those that must be kept are all incoming references. Those references that can be found by following internal references from the roots and wanted references are given by **w**. Similarly **k** gives those reachable from the incoming references which must be kept. The outgoing references which are definitely wanted are given by the set **accessible'**. Variable are recovered if they are not referred to by wanted or kept references.

This completes the specification of the leaf area's garbage collector, however some further definitions are required for the refinement. The generic function _map_ maps a set of values of some type X and a value of another type Y to a function from X to Y. This resulting function maps all values in the set to the single value.

```
[X,Y]
(_ map _) : ( ℙ X × Y ) → ( X → Y )
---
∀ x : ℙ X; y : Y | x ≠ {}
                 • dom( x map y ) = x
                 ∧ ran( x map y ) = {y}

∀ x : ℙ X; y : Y | x = {}
                 • x map y = {}
```

The specification is to be refined to a scanning garbage collector, which uses marks to control the scan. Therefore a set of marks is introduced to represent these. Three identifiers are declared to represent the three different kinds of mark which will be used.

[Mark]

not_found, found, scanned : Mark

#{ not_found, found, scanned } = 3

The predicate ensures that the three types of mark are all distinct. Other types of mark may exist in the set, but these are not used.

3.7.3 The Refinement to Code

The specification of the leaf area's garbage collector will now be refined to a language, based on Dijkstra's guarded commands [Dijkstra75], whose variables are sets and functions which correspond to the types in Z. Further refinement would be necessary to derive code expressed using the control and data structures found in the target language. In particular this would involve data refinement, such as refining sets into lists. These steps are omitted but would proceed in a similar fashion to those presented.

The specification, given by COLLECT, can can be refined to a two step process which first identifies the wanted variables and then those that must be kept in case they later prove to be accessible.

COLLECT ⊑ COLLECT1 ; COLLECT2

This refinement can only be made if the following proof obligations can be satisfied.

1. pre COLLECT ⊢ pre COLLECT1
2. pre COLLECT ∧ COLLECT1 ⊢ (pre COLLECT2)'
3. pre COLLECT ∧ COLLECT1 ∧ COLLECT2' ⊢ COLLECT[_''/_']

The first states that COLLECT1 is applicable whenever COLLECT is applicable, the second ensures that COLLECT2 can be applied in all states that result from applying COLLECT1 and the third states that the result of applying COLLECT1 and then COLLECT2 satisfies the specification of COLLECT. The proofs of these, and all other propositions made in this section, are relatively straightforward and are given in Appendix D.

The following specifications are proposed for COLLECT1 and COLLECT2:

COLLECT1

```
keep, keep', wanted, accessible', foundin' : ℙ Ref
```

```
wanted ⊆ dom( incoming )
keep = keep'

  accessible' = outgoing ∩ w
  foundin' = w \ outgoing
where
  w ≙ ( internal ; contains )* (( roots ∪ wanted ))
```

COLLECT2

```
keep, foundin, accessible, accessible' : ℙ Ref
recover' : ℙ Var
```

```
keep ⊆ dom( incoming )
accessible' = accessible

  recover' = vars \ (incoming ∪ internal)(( foundin ∪ k ))
where
  k ≙ ( internal ; contains )* (( keep ))
```

The specification of the first sequent, COLLECT1, can be further refined into an initialising step and a scanning phase, using marks to control the scan.

COLLECT1 ⊑ INIT1 ; SCAN1

The proof obligations that must be satisfied are similar to those of the previous step.

4. pre COLLECT1 ⊢ pre INIT1
5. pre COLLECT1 ∧ INIT1 ⊢ (pre SCAN1)'
6. pre COLLECT1 ∧ INIT1 ∧ SCAN1' ⊢ COLLECT1[_''/_']

The following specifications are taken for the two steps:

INIT1

```
marks, marks' : Ref ⇸ Mark
wanted, accessible', keep, keep' : ℙ Ref
```

```
keep' = keep
dom( marks ) = dom( internal )
marks' = ( dom( marks ) map not_found )
              ⊕ ( (roots\outgoing ∪ wanted) map found )
accessible' = roots ∩ outgoing
```

SCAN1

```
marks, marks' : Ref ⇸ Mark
accessible, accessible', keep, keep', foundin' : ℙ Ref
```

```
  accessible' = accessible ∪ (refs ∩ outgoing)
  marks' = marks ⊕ (refs\outgoing map scanned)
where
  refs ≙ marks⁻¹;( internal ; contains )* ⦇ {found} ⦈

keep' = keep

ran( marks ) ⊆ { not_found, found }
ran( marks' ) ⊆ { not_found, scanned }
dom( marks ) = dom( internal )
dom( marks' ) = dom( marks )

dom( marks' ▷ {scanned} ) = foundin'
```

The scanning phase can now be refined into a initialised loop construct. The initialising step establishes an invariant condition, which relates the state of the variables before the execution of the constuct to their state after each iteration. The simplest form of the construct is used. This has only one guard and so effectively eliminates the non-determinism.

SCAN1 ⊑ INV ; **do** GUARD → BODY **od**

Total correctness of this refinement is assured by satifying the following propositions:

7. pre SCAN1 ⊢ pre(INV ∧ ¬ GUARD')
8. pre SCAN1 ∧ INV ∧ ¬ GUARD' ⊢ SCAN1
9. pre SCAN1 ∧ INV ∧ GUARD' ⊢ (pre BODY)'
10. pre SCAN1 ∧ INV ∧ GUARD' ∧ BODY' ⊢ INV[_''/_']
∧ bound(marks'')<bound(marks')
11. pre SCAN1 ∧ INV ∧ GUARD' ⊢ bound(marks') ≥ 0

The first states that there is a state in which the loop will terminate. That is the invariant holds but the guard does not. The second ensures that the specification of SCAN1 is satisfied after the loop has terminated and the third that the loop body must be valid in all states in which it may be applied. Termination is ensured by the fourth proposition, which states that there is a measure that decreases for each iteration. The last proposition states that the measure is applicable in all required states.

The following schemas specify the parts of the loop construct:

INV

$$
\begin{array}{l}
marks,\ marks' : Ref \pfun Mark \\
accessible,\ accessible',\ keep,\ keep' : \mathbb{P}\ Ref \\
\hline
accessible' = accessible \cup r \cap outgoing \\
\textbf{where} \\
\quad r \triangleq marks^{-1};(marks'^{-1}(\{scanned\}) \lhd internal;contains)^{*}(\{found\}) \\
\\
marks'^{-1}(\{not\_found\}) \cap marks'^{-1};internal;contains(\{scanned\}) = \{\} \\
marks^{-1}(\{found\}) \cap marks'^{-1}(\{not\_found\}) = \{\} \\
dom(\ marks'\ ) = dom(\ marks\ ) \\
\\
keep' = keep
\end{array}
$$

GUARD

$$
\begin{array}{l}
marks : Ref \pfun Mark \\
\hline
\exists\ next : Ref \bullet marks(\ next\ ) = found
\end{array}
$$

BODY

$$
\begin{array}{l}
marks,\ marks' : Ref \pfun Mark \\
accessible,\ accessible',\ keep,\ keep' : \mathbb{P}\ Ref \\
\hline
keep' = keep \\
\\
\exists\ next : Ref \mid marks(\ next\ ) = found \bullet \\
\quad marks' = marks \\
\qquad \oplus\ (\ (marks^{-1}(\{not\_found\}) \cap \\
\qquad\quad contains(\{internal(next)\})) \mapsto found) \\
\qquad \oplus\ \{next \mapsto scanned\} \\
\quad accessible' = accessible \\
\qquad\qquad \cup\ (contains(\{internal(next)\}) \cap outgoing)
\end{array}
$$

$$
\begin{array}{l}
bound : (\ Ref \pfun Mark\ ) \to \mathbb{N} \\
\hline
\forall\ m : Ref \pfun Mark \bullet bound(\ m\ ) = \#\ m \rhd \{scanned\}
\end{array}
$$

Figure 3.7a shows the refinement steps which have been presented in this section. Further refinement would produce code for the initialisation step and the second phase of garbage collection.

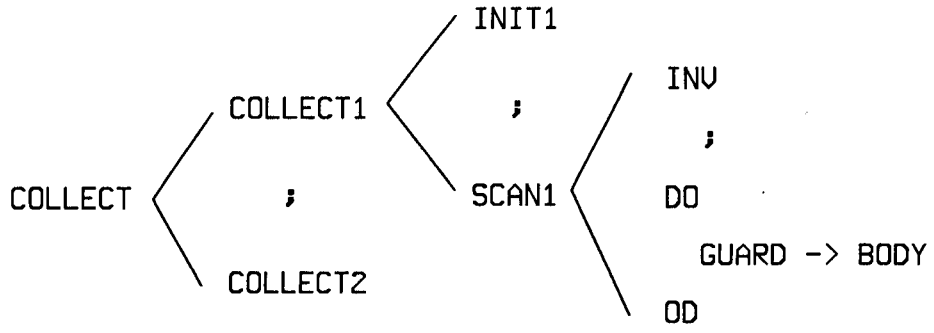


Fig3.7a: Part of the specification for a leaf area's garbage collector is refined to a guarded command language program.

Figure 3.7b shows the guarded command program which is the final result of the refinement of COLLECT1.

```
INTI1;
do
  ∃ next : Ref • marks( next ) = found
→
  choose next : Ref | marks( next ) = found
     marks := marks ⊕ { next ↦ scanned };
     do
       ∃ r : Ref | r ∈ contains( next ) • marks( r ) = not_found
     →
        choose r : Ref | r ∈ contains(next) ∧ marks( r ) = not_found
           marks := marks ⊕ { r ↦ found }
     od
od;
COLLECT2
```

Fig3.7b: The guarded command language program resulting from further refinement of the inner loop.

The complete garbage collector would be developed in much the same way as this example portion. However in this example only the control structures were refined. In practice it will be necessary to refine the data structures used as well. For example, in the garbage collector of an internal area, the actions of the get_wanted and get_keep functions must be refined into information held in tables. These would be refined into centralised or distributed indirection tables as appropriate.

It will also be necessary to specify the interaction between elements of the distributed system, in particular the distributed termination protocols described in section 4.4. For this the language Z does not seem appropriate, and it may be that the complete garbage collector will be specified using some Z and some CSP [Hoare85].

3.8 Summary

This section has suggested that recursively structuring a heap into separate areas, which are garbage collected largely independently, will give a better utilisation of memory than simply garbage collecting the heap as a whole. Those areas which are divided into more areas are garbage collected by combining the effects of garbage collecting these offspring areas, rather than performing a separate "global scan" such as that described by [Ali&Haridi85]. The structuring does not alter the user's view of the heap, but should allow some inaccessible variables to be recovered more quickly compared with the use of an unstructured heap.

The garbage collection algorithm proposed for use with the recursively structured heap is of the marking variety. It was presented initially as a recursive algorithm and this was informally refined to a more practical algorithm in which parallel processes independently garbage collect the leaf areas. These processes coordinate their actions, using structures which are recusively created, to garbage collect the higher level areas.

The garbage collection processes and the computation processes using the heap must cooperate to prevent accessible variables being marked inaccessible and to ensure that each garbage collection cycle completes. This would add some synchorisation overhead to a system where computation and garbage collection are executed on separate processors. However in a single processor implementation their interaction would be controlled like coroutines, with no extra overhead.

In view of the importance of the correct operation of the garbage collector, it is suggested that the code should be derived from the specification using rigorous program development techniques. To illustrate this a part of the garbage collector for leaf areas was specified, using Z, and refined into a guarded command language program. From this the refinement step to executable code is relatively straightforward. This small exercise in rigorous development has shown that such production of a complete garbage collector is possible, though it will require considerable effort.

The important difference between the new algorithm and other distributed garbage collectors is that it guarantees to recover all inaccesible variables, without needing to move them between areas. In addition the heap is recursively structured, whereas other distributed garbage collectors only work for two level heaps. Also, unlike other distributed scanning garbage collectors, the scan is effected by combining the results of the lower level scans rather than with an extra scanning activity.

4. Practical Implementation

This chapter discusses the practical implementation of the garbage collector described, in abstract form, in chapter three.

4.1 Indirection Tables

A number of practical problems arise when it comes to implement the garbage collector described in chapter three.

First, within one computer of a distributed system, references are likely to be small, occupying one or perhaps two words. However references to variables stored in other computers are likely to be much larger, because they must include the network address of the computer as well as the variable's address within it. It would be inefficient to make all references large enough to accomodate inter-computer references, because the majority will refer to variables in the same computer. However, it would be convenient if all references were the same size, as this would allow software to manipulate them easily.

Secondly, storage would be wasted if global garbage collection information were reserved for all variables, since the majority of them will only be referenced locally.

Thirdly, each computer's local garbage collection is relatively autonomous, and so is any store compaction that takes place. However, if a variable which is referenced from outside the computer is compacted, all references to it in the distributed system as a whole must be updated. This could involve an extremely large search.

Each of these problems can be solved by using indirection tables. An indirection table for variables referenced by other computers would contain the variable's local address and its global garbage collection information. This solves the last two problems. An additional indirection table for outgoing references solves the first problem. This would contain the network address and local address of the referenced variable.

A further advantage of an outgoing indirection table is that the garbage collector's network traffic can be reduced. If a computer has many references to a variable in another computer, the garbage collector will send a mark message for each of the references. If all the references use the same outgoing entry to refer to the variable, a flag on the entry could indicate that the mark message has already been sent. Thus only one message would be sent per variable rather than one per reference. The flags would be reset during the garbage collector's recovery phase.

A further possibility is that one incoming indirection table entry can be referred to by outgoing entries in many different computers. Not only does this save space in the incoming indirection tables, but leads to the reference counting optimisation described in section 4.6, although this does lead to complications with network partitioning, as discussed in section 4.7.

4.2 Compressing the State Information

If a central indirection table holds information about several levels of inter-area references, it is possible to compact the state information considerably. Each inter-area reference may pass through several levels of area, and state information is required at each level. However, it can be seen from the algorithm that if a reference is wanted at a high level, it is treated as wanted at all lower levels.

It is sufficient to record only the level of the highest level reference referring to a variable, along with the highest level at which it becomes found or scanned. The central indirection table need therefore only contain one entry per variable, at the expense of two extra fields to record the level of reference and the level at which it is found or scanned. Not only is this likely to save space, but it should also speed up addressing variables using high level references. Instead of several indirections, through each level's indirection table, only one indirection is taken regardless of the level of the reference.

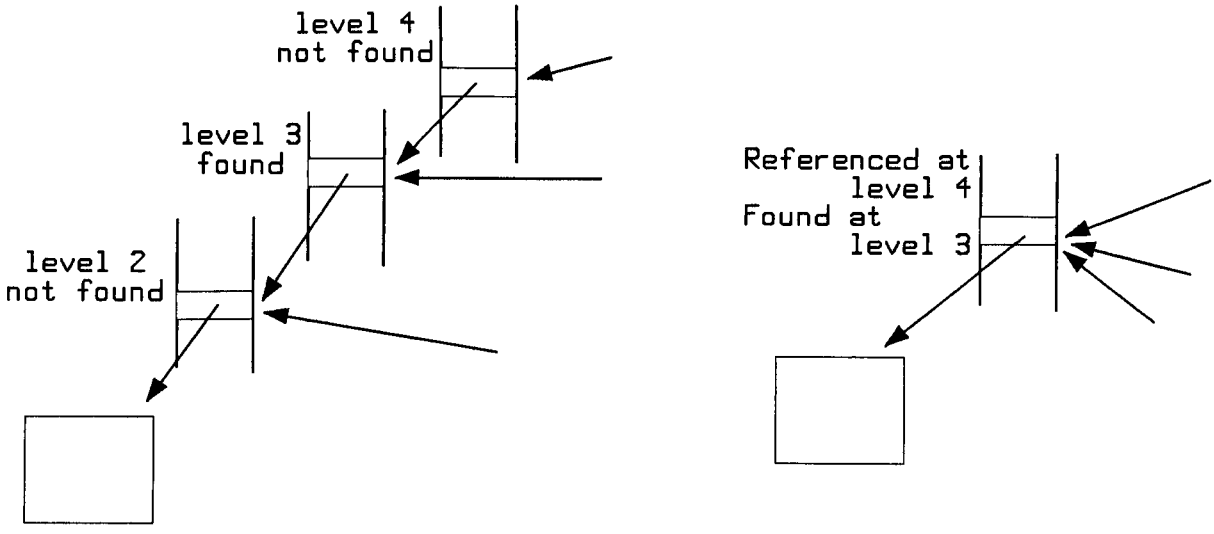


Fig4.2: State information for the garbage collection of several levels.... .....can be compacted.

The use of logical areas, described in section 4.5, is an important example of garbage collection with centralised control and benefits from using the compression technique.

4.3 Centralised vs. Distributed Control

If control of the garbage collection of an area is centralised in one computer, the state information for all internal and incoming references can be held in one central indirection table. In its simplest form the table is an array. For each reference the table records its state in the garbage collection, that is whether it is not found, found, scanned or new, and some addressing information. This comprises the address of the offspring containing the variable referred to, and some site dependent address, which may be a further index into an indirection table or an actual physical address.

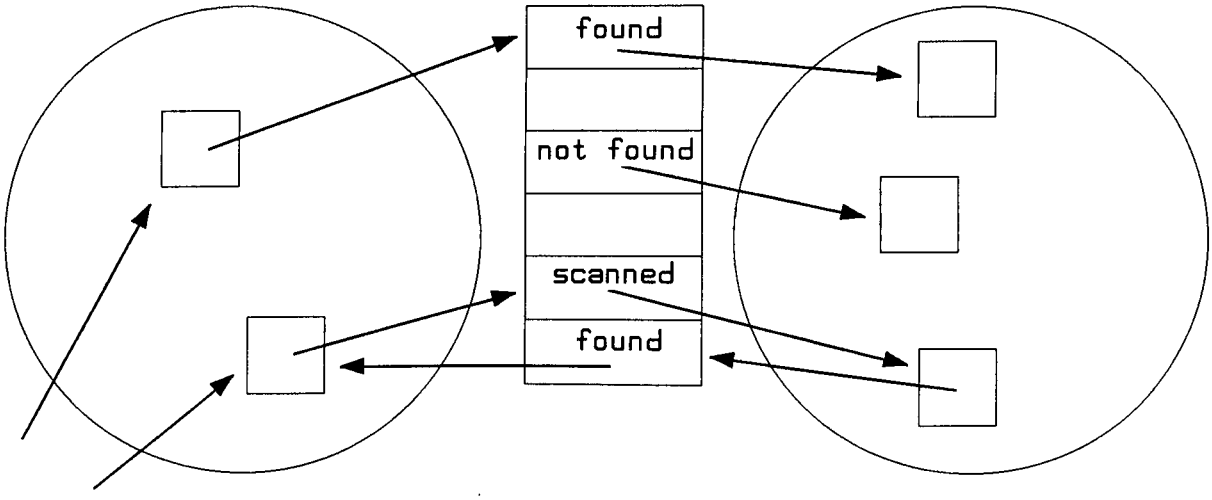


Fig 4.3a An indirection table held centrally allows garbage collection to be controlled easily but can become a bottleneck.

A gateway connecting two networks, each of which is garbage collected as an independent area, is a suitable place for a centralised controller. All reference information must pass through the gateway, so garbage collection information is easily maintained. In such a configuration, extra processes run in the gateway to handle the higher level garbage collections, but only as controllers, they do not themselves perform any scanning. If, however, the networks are connected by more than one gateway, the gateways must cooperate and the benefits of centralised control are lost.

The offspring garbage collectors communicate with the centralised controller to establish the wanted and keep sets for the scans. These may be obtained piecemeal. Each time the offspring requests an additional piece a simple scan can be used to find more members of the set. When the offspring find accessible outgoing references, they inform the centralised controller, which marks the reference accordingly.

With control of the garbage collection centralised, it is relatively straightforward to detect termination of the scanning phases, using much the same technique as given in the abstract algorithm of chapter 3. However, the problem with this approach is that the centralised controller becomes a bottleneck in the system.

For a loosely coupled network in which each computer is a separate garbage collection area, control of the garbage collection needs to be distributed amongst all of the computers, to avoid the bottleneck of a centralised controller. Each computer must record information about the references which it holds for variables in other computers, and about variables it holds which are referenced by other computers.

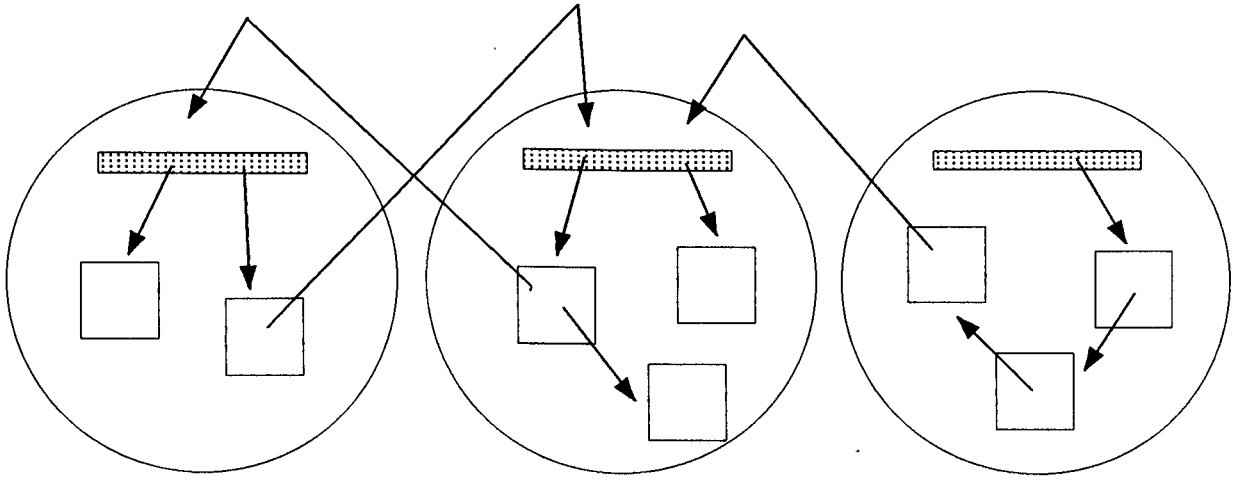


Fig4.3b An indirection table distributed amongst the computers in the system speeds access, but synchronisation is more difficult.

To do this each computer contains part of the indirection table of the system. The entry for a variable which is remotely referenced is held in the computer which contains the variable. Within each computer, the indirection tables for each level may be compressed using the technique described in section 4.2.

With the reference information distributed amongst the sites interested in it, it is easily and speedily accessed with no centralised controller causing a bottleneck. However there is a problem in determining when a garbage collection scan has completed, since the required information is distributed. This is dealt with in the next section.

4.4 Distributed Termination Detection

The general problem of distributed termination detection is as follows. Each computer, or node, in the system is independently evaluating part of some computation. In the course of its computation, a computer may send a computation message to another, causing the receiver to perform more work. However, when a computer has completed its task, it cannot send any computation messages. Hence the computation exhibits a stability property, in that once all parts of the computation are completed, it does not start up again. The problem is for the computers to detect when the computation as a whole has finished.

In the scanning phase of a distributed garbage collection, each garbage collector has finished its work when it has no more **found** variables that require scanning. While it is scanning, the garbage collector may discover a reference to a variable in another area. It will send a mark message to that area's garbage collector, which may cause it to continue its scanning. Once there are no more **found** variables in the area, no mark messages may be sent and the scanning has finished. This is the state the garbage collectors must detect, however it is sufficient for only one of them to detect it, because it is then able to notify all the others by broadcasting.

A solution to the problem of distributed termination detection is offered by [Francez80]. One node is distinguished from the others as the controller, which is the node that detects termination. It sits at the root of a spanning tree which covers each node in the distributed system. For the termination detection algorithm, nodes only communicate with the node that is their parent in the spanning tree and with their children.

The algorithm starts with the controller sending a wave of 'test' messages down the tree. On receipt of such a message, an internal node that has not completed its part of the distributed computation replies immediately with a 'busy' message. If a node has completed its computation it propagates the 'test' message to all its children and waits for replies from them. While it is waiting it freezes, that is it refuses to take part in the distributed computation. If any other node attempts to communicate with it, that node must wait. Leaf nodes which have completed their computation return an 'idle' message.

Once a frozen node has received replies from all its children it returns either a busy message, if any of the replies are busy messages, or an idle message, if all the replies are idle messages. Eventually the controller receives a 'busy' or an 'idle' message. If an 'idle' message is received, the controller can conclude that the distributed computation has completed. However, if a 'busy' message is received the controller must propagate a wave of 'unfreeze' messages down the tree to allow the frozen nodes to continue.

The problem with this algorithm is that the distributed computation is frozen while termination detection takes place. In the case of distributed garbage collection this could introduce pauses in execution, which, it has been argued, are highly undesirable.

An improved algorithm is offered by [Francez&Rodeh82]. [Topor84] develops the same algorithm, but in a rigorous way. This algorithm achieves termination detection without freezing the distributed computation. The algorithm starts by propagating a wave of 'idle' messages up the tree. Leaf nodes send an 'idle' message to their parent when they become idle. An internal node propagates the 'idle' message when it has received messages from all its children and is itself idle. When a node sends the 'idle' message, it sets a flag, 'remained_idle', to true. If an idle node receives a computation message that causes it to start computing again, the remained_idle flag is set to false.

Once the controller receives a 'idle' message, and is itself idle, it knows that all the 'remained_idle' flags have been set. It now propagates a wave of 'test' messages down the tree. Once this wave reaches the leaves, another wave of messages is returned which reports on the state of the 'remained_idle' flags. In fact this wave is combined with the first wave. It returns either a 'busy' or 'idle' message depending on the state of the 'remained_idle' flags and sets the flags back to true.

A similar solution is proposed by [Dijkstra et.al.83], though this arranges the nodes in a ring. This arrangement effectively dispenses with the wave of 'test' messages by propagating one message around the ring. The controller waits until it is idle and then sends an 'idle' message around the ring. On receipt of the message, a node propagates either a 'busy' or an 'idle' message depending on the value of its 'remained_idle' flag and whether it is itself still busy. The flag is reset when the message is propagated. When the controller receives an 'idle' message it can conclude that the distributed computation has completed.

With the nodes communicating via a spanning tree, at best 2(n-1) messages are required to detect termination, while the ring algorithm gives a best case of n messages. However with the ring arrangement, termination detection could take longer to complete because messages are not sent in parallel. Using a spanning tree at worst 2(n-1) messages are needed to detect that the computation has not yet completed, while with a ring n are always required. If may therefore prove better to use a spanning tree if termination detection usually fails.

A possible compromise would be to span the nodes with a lattice like structure, where the top and bottom are both the controller. This allows messages to be propagated in parallel, giving the advantages of the spanning tree algorithm while avoiding the wave of test messages it requires.

The algorithms just described allow a single controller to detect termination. However, for the distributed garbage collector it would be desirable if any of the nodes can detect termination of the scans. This would make each system identical which would avoid configuration problems. One possible way of achieving this is if each node has 'n' flags, and 'n' termination algorithms proceed in parallel, each controlled by a different node. However this causes many more messages to be sent across the network.

The number of messages can be reduced if a node which is trying to detect termination communicates with a busy node. The node may now cease trying to detect termination because it can rely on the busy node doing this when it becomes idle. This is essentially the aim of algorithms described by [Rana83] and [Arora et al.87]. The former requires the use of synchronised clocks, which are expensive to maintain, but the latter presents a simpler solution.

4.5 Logical Areas

The parallel-recursive algorithm which is presented by this thesis has been developed for garbage collecting distributed heap stores. However, it may also be used within a single physical memory, where the store is divided into logical areas. Here, each variable is assigned to one logical area, though may be physically allocated anywhere in the store. References may be stored freely in any variable, thus the user perceives no difference between intra-area and inter-area references.

Such a system with logical areas may be used to provide a limited degree of isolation between the users of a shared heap. If the users are each given a different logical area in which to allocate their variables, garbage collection is largely performed on a per user basis. The time used garbage collecting a logical area can then be taken from the owning user's cpu time budget. In this way users who produce large amounts of garbage, and consequently require a larger percentage of garbage collector activity, cannot deprive other users of cpu time.

Similarly, each logical area could be given a physical store budget. When new variables are allocated, the budget is decreased and when the garbage collector recovers inaccessible variables it is increased. The user is prevented from allocating a variable if it would cause the budget to expire. In this way, one user of the shared heap cannot allocate all the available store, to the detriment of other users.

This ability to control the usage of the shared heap is most important in multi-user systems. It is also relevant in systems where availability of service is of concern. By allocating different sub-systems to different logical areas, it is possible to contain the damaging effects of errant programs which consume store and, indirectly through excessive garbage collection, cpu time.

The use of logical areas to control the activities of users is to be used in the SMITE capability computer, both in its main heap store and capability based, write once backing store [Wiseman88]. The computer is being developed for computer security applications, and logical areas are required to provide protection from denial of service threats.

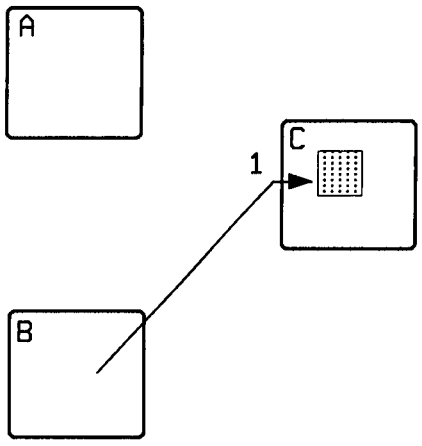
The use of logical areas within one computer's memory is another example garbage collection with centralised control. Here all the information about references between areas is resident in one place, making distributed termination detection unnecessary.

4.6 Reference Counting

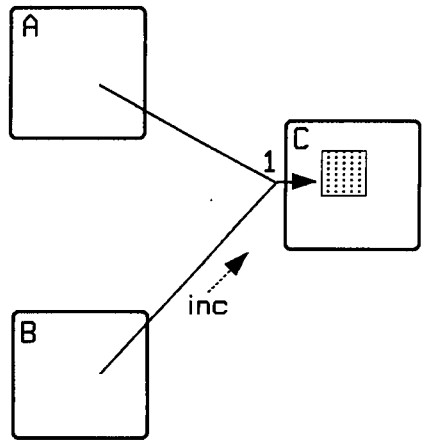
Indirection tables were introduced in section 4.1 as a solution to a number of problems. However their use presents the opportunity to optimise the garbage collection by using reference counting.

Each entry in an incoming indirection table would record the number of times it is referenced by an outgoing entry. When an outgoing entry is no longer required, the count of the incoming entry it refers to is decremented. If a reference count reaches zero then the incoming entry is no longer required and may be recovered.

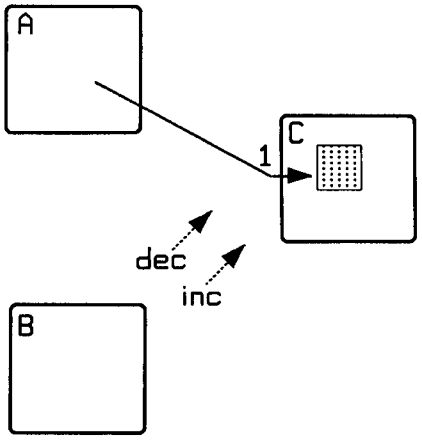
In systems where the majority of garbage is not cyclic, reference counting should prove to be a worthwhile optimization.



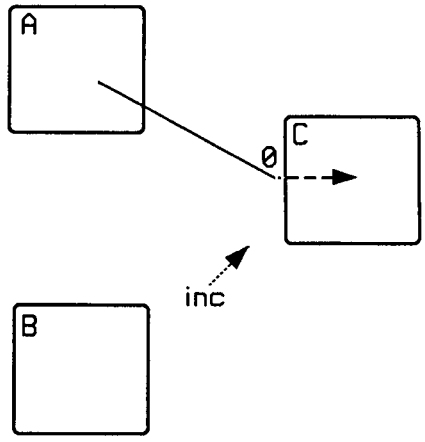
1 Initially computer B has a reference for a variable in computer C.



2 The reference is copied to computer A, so an increment message is sent to C.



3 Computer B discards all its references to the variable, so its outgoing indirection table entry is recovered and a decrement message is sent to C.



4 If the decrement overtakes the increment, the reference count will drop to zero and incoming indirection table entry will be removed, leaving the reference in A dangling.

Fig4.6: Problems can arise when references are moved between computers.

In a distributed environment care must be taken to ensure that undelivered requests to increment a count are taken into consideration before recovering any entries. Problems arise if messages sent across the network are reordered or when references are moved to other computers, as shown in figure 4.6.

These problems can be overcome by using the weighted reference count mechanism proposed by [Watson&Watson87]. This is where each reference contains a weight and the variable contains a count which is the sum of the weights. When a reference is copied, its weight is split between itself and its copy. In this way a variable's reference count need never be incremented.

This scheme is ideal for the proposed reference count optimisation because the weights are held in the outgoing indirection table entries, and so their size is not a burden.

The method proposed by Watson and Watson for handling the case when a reference with a weight of one is copied, presents some difficulty. Their solution is to introduce an extra level of indirection when this occurs. A new variable is created which contains the reference to be copied. Then references to this variable are copied instead.

A more appropriate solution to this problem is to abandon reference counting for that particular incoming indirection entry and rely on the scanning garbage collector to recover it. This is achieved by setting the weights of the original outgoing entry and the copy to zero. The reference count in the incoming entry is now greater than the sum of the weights in the outgoing entries, and so it will never drop to zero. Note that, whenever an outgoing entry with a weight of zero is discarded, there is no need to send a decrement message.

4.7 Fault Tolerance

Any algorithm designed for use in a distributed system must consider the effects of failures. Communications within distributed systems cannot be considered completely reliable. Messages may be corrupted or lost completely, they may be delayed, delivered out of order or delivered more than once. Individual computers within distributed systems may crash, losing the contents of their memory, or may be temporarily isolated from others due to network failures. A taxonomy of such faults is given by [Ezhilchelvan&Shrivastava85].

4.7.1 Message Loss and Corruption

It is assumed that corruption of messages can be detected by adding suitable checksums and redundancy. Corrupt messages can then be discarded and treated in the same way as lost messages. However, other forms of failure can still be encountered.

Such communications failures affect both the distributed garbage collection and distributed computation that uses the heap. However, only the problems of the former are considered here. Communication between the garbage collectors takes place at three different times: sending a marking message, detecting termination of the scan of an area and updating reference counts, if they are used.

Marking messages, sent when a garbage collector discovers an outgoing reference is accessible, may not be delivered. If no further action is taken, the variable may erroneously be identified as garbage. However, if the marking message is sent more than once, no problem arises. The first message causes the variable to be marked as wanted, so subsequent messages have no effect.

Therefore, to avoid the problem of message loss, a simple retry scheme is sufficient. The recipient of a marking message acknowleges it by returning a special **ack** message. If the sender fails to receive an ack before some timeout period expires, it sends the mark message again. Note that loss of the ack message causes no problem, because the mark message may be received more than once without ill effect. One possibility is that the timeout period could be up until the end of the scan, which would allow plenty of scope for blocking together multiple ack messages.

If a marking message is delayed for a long time, it may arrive during the next scan phase. If the information it carries can be validly interpreted in the context of the current scan phase, the mark can proceed. At worst this will cause some inaccessible variable to be marked as wanted. However, if the information does not make sense, the message can be safely ignored. The only danger is if the information is interpreted in an invalid way, for example marking a variable which no longer exists. This must be avoided, because the space used by the marks of the old variable may now be used as part of the contents of a new variable. The recipient garbage collector must ensure that the marking messages it receives are applicable to the current context. Hopefully this is simply a matter of checking that the indirection table index is in range, however it may be necessary to include large sequence numbers in all mark messages, indicating which scan they form part of.

If communication faults affect the termination detection algorithm, the garbage collection scan may never be terminated or may even terminate early causing accessible variables to be discarded. Termination detection is achieved by passing tokens around the nodes. These will need to be repeatedly sent until a positive acknowlegement of their arrival is obtained. However, they must be delivered exactly once to avoid the problem of premature termination detection. The "orphan killing" technique of [Panzieri&Shrivastava85] can be employed to ensure this.

Messages which decrement the reference counts for inter-area references also suffer from the various forms of failure. However, reference counting is only suggested as a performance improvement to the main scanning algorithm. Therefore, if a reference count is actually too high no problem is caused. This happens naturally with cyclic structures and when, with the weighted reference count scheme, the weights in the outgoing indirection table entries drops to one.

Decrement messages must not be delivered more than once, however it is safe for them to be completely lost. If the distributed system can guarantee that messages are never duplicated, it is sufficient to simply send the decrement message once and assume it arrived safely. However, if message duplication is possible, steps must be taken to ensure that duplicate messages are ignored.

Perhaps the simplest solution to the problem of duplicate decrement messages is to sequence number all decrement messages which travel from one computer to another. This would allow duplicate messages to be discarded, though it may also discard some out of order messages.

4.7.2 Network Partitioning and Computer Crashes

A further problem of distributed communication is distinguishing between network partitioning, where failures in communications equipment prevents one part of the distributed system communicating with the other, from crashes of individual computers. In the former case, variables still survive and may become accessed in future if the network mends. In the latter case the variables are destroyed, and hence any variables they refer to on other machines may become garbage.

If a computer cannot communicate with another, it cannot find out which of the variables that have been referenced by that computer are still required. It could keep all these variables, and any accessible from them, in case communcations are reestablished and the other computer still has references to the them, or it could assume that the other computer has crashed and therefore has no references to the variables.

In either case it is necessary to ascertain which variables may be referenced by the other computer. This is either to preserve them in case the connection is re-established or to discard them if the other computer is assumed to have crashed. In order to do this, a one to one correspondence between outgoing and incoming indirection entries must be preserved. This unfortunately has a nugatory effect on the reference counting optimisation described in section 4.6.

When a computer holding externally referenced variables crashes, the variables are lost. However the references held in the other computers must not be left dangling. This can be achieved by recording the time at which a computer is initialised. This time is made a part of all references to variables stored in the computer. When an external reference is used a check is made to see if the time in the reference is the same as the time that the system was initialised. If not the system must have crashed and so the reference is invalidated.

A technique that uses this kind of approach is described in [Mancini&Shrivastava87]. Here a fault tolerant reference counting garbage collector for a distributed system is integrated into an orphan detection and killing system for remote procedure calls.

4.8 Weak References

Some systems have two forms of reference, weak and strong. They behave in exactly the same way with regards to addressing variables, but only strong references protect a variable from garbage collection. If a variable is only referenced by weak references, all the references can be changed to nil and the storage occupied by the variable recovered.

4.8.1 Applications for Weak References

Weak references are used in some LISP systems [Lieberman&Hewitt83] and in the Flex capability computer [Fosteret.al.79]. They can be used to maintain information about an object for as long as it is required, without causing the object to be permanently accessible. Figure 4.8a gives an example. Here a table is maintained recording information about various objects; three are shown. $Object_1$ is accessible by some strong references, and so the weak reference, shown as a dashed line, remains. However $object_2$ is not referenced by an accessible strong reference. Therefore the weak references can all be turned to nil and the object can be recovered. When the table is scanned, the pointer to $object_2$ will be found to be nil, and so the information can be discarded by removing it from the table.

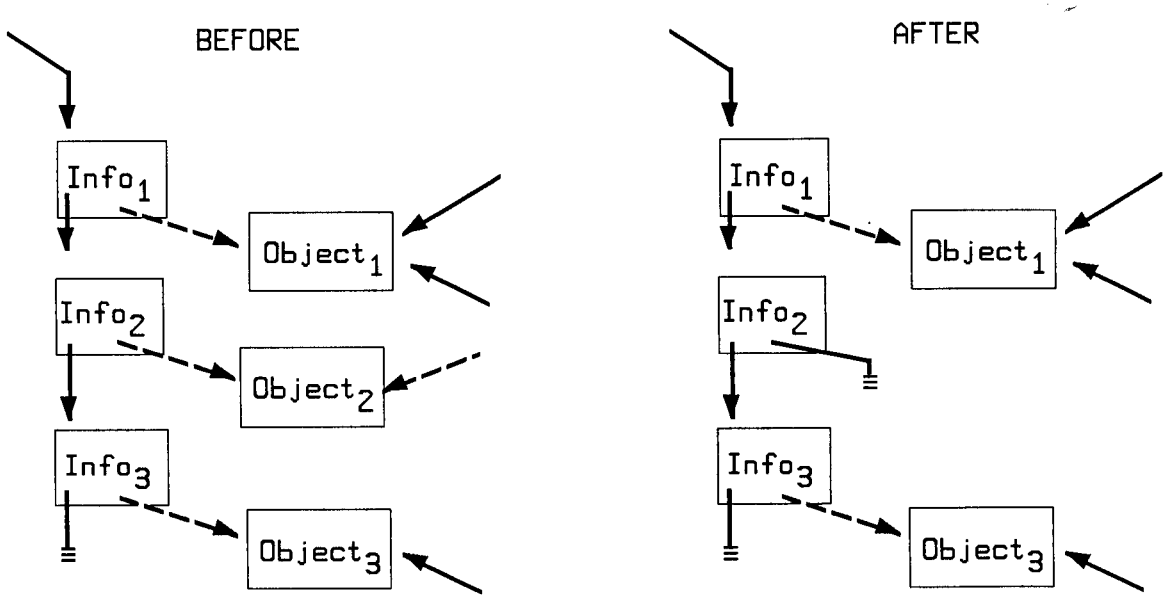


Fig4.8a: Weak references do not protect an object from garbage collection. When all the accessible references to a variable are weak, they turn to nil and the object is recovered.

In the Flex distributed system [Foster&Currie86], weak references are used to maintain information about inter-computer references. A similar scheme is used by [Vestal87]. A list of outgoing references is maintained for garbage collection purposes. To detect when the entry for an outgoing reference is no longer required, the list elements refer to the variable that represents the remote object with a weak reference. The users refer to this variable, probably with strong references, rather than the list element.

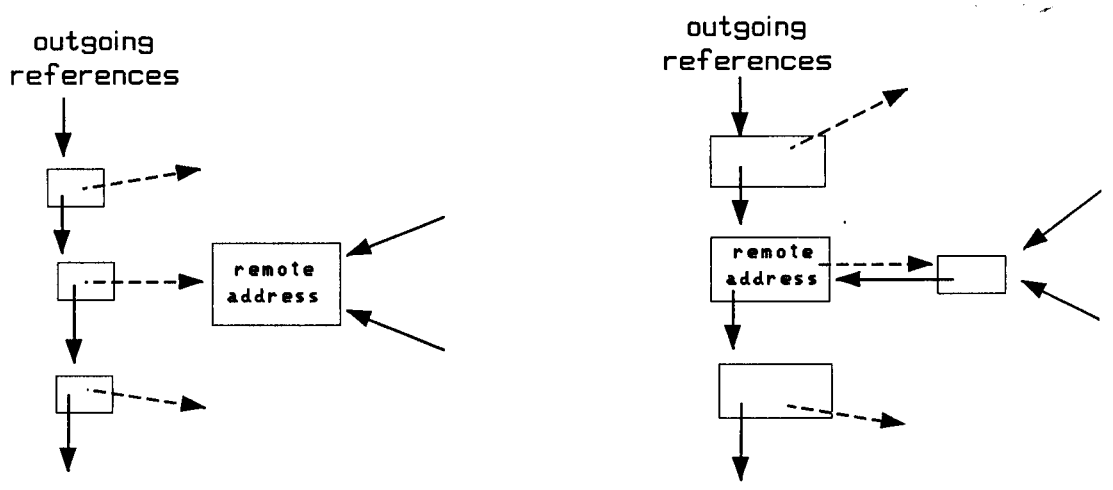


Fig4.8b: The list of outgoing references uses shaky references to allow them to be garbage collected. The remote address may be stored in the variable which is the local representation of the remote object or in the list element.

The remote address of the object may be stored in either the list elements or the variable which the users refer to, as shown in figure 4.8b. In the former case the address is available after the object is no longer referenced, and so can be used, for example, to decrement a remote reference count. In the latter case the address is not available, but the address is available without extra indirection.

Weak references may also be used to remove redundant processes. These arise from eager evaluation in applicative systems [Moor82]. The process refers to a variable in which it will place its result, as shown in figure 4.8c. The process' client also refers to this variable, but only for as long as the result is relevant. The computer's scheduling queue uses weak references to refer to processes, so it does not keep them alive. A strong reference to the process must be kept in the result variable. This keeps the process alive, but only for as long as the result variable remains alive. If the result of a process becomes irrelevant the user discards all references to the result variable, therefore it and the process become garbage. The scheduler will notice a nil reference in its scheduling list, which it will remove.

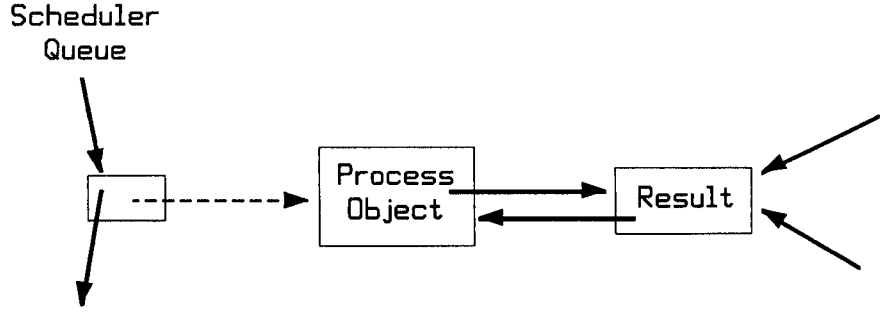


Fig4.8c: The scheduler refers to processes with weak references so a process is garbage collected away if it is no longer relevant.

Special purpose garbage collectors for eliminating redundant processes have been proposed by [Baker&Hewitt77], [Grit&Page81] and [Hudak&Keller82]. These all involve making processes a special case during garbage collection. However it is unclear whether they achieve superior performance over a system using weak references.

4.8.2 Garbage Collection of Weak References

Before describing how weak references are treated during garbage collection, it is worth noting that their introduction violates the first essential criterion given in section 2.1.1. This is because the garbage collector can legitimately alter the shape of the accessible structure, by changing weak references to nil when no strong reference to the variable is accessible. Thus the safety criterion has to be relaxed slightly.

The garbage collector must be able to determine when a variable is referred to only by weak references. To this end, the state flag of each variable is allowed to take the value **weakly found**, in addition to **not found**, **found**, **scanned** and (perhaps) **new**.

When the garbage collector finds a weak reference, while searching a **found** variable for references, it examines the state of the variable it refers to. If the variable is marked as **not found**, this is changed to **weakly found**, otherwise it remains unchanged. When the search finds a strong reference to a **not found** or a **weakly found** variable, it marks the variable as **found**. This is summarised in figure 4.8d.

On finding	change to	Weak Reference	Strong Reference
not found		weakly found	found
weakly found		weakly found	found
found		found	found
scanned		scanned	scanned
new		new	new

Figure 4.8d: Effect on finding a reference to a variable.

At the start of the garbage collector's recovery phase, variables are marked either **not found**, **weakly found** or **scanned**. Those marked **not found** are inaccessible and are recovered. Those marked **scanned** are probably accessible and are kept, with their mark being changed back to **not found** ready for the next garbage collection cycle. Those marked **weakly found** are only referenced by weak references.

The garbage collector must change all references to the **weakly found** variables to nil. However, to avoid an extensive search for them, a tombstone [Lomet75] is erected on the site of the variable. This can be done by flagging either the variable itself or its indirection table entry to indicate that the variable has become garbage. At this point it may be possible to recover some or all of the storage of the variable.

Whenever a weak reference is used by the computation, a check must be made to ensure that the variable it refers to has not become garbage and been replaced by a tombstone. If it has, the reference is immediately changed to nil and the access is denied. If the variable is still alive, the computation will mark it to indicate that it is still accessible.

While the garbage collector is searching accessible variables for references, it checks any weak references it finds to see if they refer to tombstones. If so the weak reference is replaced by nil. Once the scan phase has been completed, all weak references to tombstones will have been replaced by nil. Therefore the existing tombstones are no longer needed and can be recovered by the recovery phase.

Thus the basic scanning garbage collector is readily extended to cater for weak references. The garbage collector described in chapter 3 is similarly extended, allowing weak references to be used in a uniform way throughout a distributed system.

4.9 Summary

This chapter has considered the practical implementation of the garbage collector proposed in chapter 3. The use of indirection tables for inter-computer references is proposed as this has a number of advantages. In particular it is possible to compact an area independently of all others. A technique has been given for compressing the information in a hierarchy of indirection tables, if these are held centrally. This not only saves space but speeds accesses to external variables.

It has been shown that garbage collection may be controlled either centrally or in a distributed fashion. With the latter there is the problem of reaching consensus, in a distributed environment, about the completion of each phase of garbage collecton. However, some techniques give in the literature have been examined and discovered to be entirely suitable.

Some consideration has been given to extending the idea of a recursively structured heap to the memory of an individual computer. This could be used to independently garbage collect areas used by separate users. Thus those users who create most garbage have to spend more time garbage collecting their own area, without affecting users who create little garbage.

The use of reference counting as an optimisation has also been considered. This allows externally referenced inaccessible variables, which are not part of a cyclic structure, to be recovered more quickly. The use of indirection tables for collecting together outgoing references is suggested as a way of greatly reducing the reference count traffic. It is proposed that the weighted reference count scheme be used to overcome problems of maintaining the reference counts in a distributed environment.

The fault tolerant aspects of the garbage collector have been addressed. The most serious problem is in resolving whether an incommunicative computer in the system has crashed or the network has partitioned, since different actions are required in each case.

A further aspect considered in this chapter, is the use of weak references. A number of uses for these are described, including removing redundant processes, and a simple extension to the garbage collection algorithm has been given to cater for them.

5. Performance Analysis & Comparisons

This chapter presents an analysis of the performance of the new incremental garbage collector for a recursively structured heap. First a simplified model is developed which shows the importance of a measure called remoteness in determining the memory utilisation that can be achieved. Secondly, experimental results are presented which suggest that the use of a structured heap and the new algorithm does give improved utilisation of memory over the use of an unstructured heap and a simple garbage collector.

To study the behaviour of the recursively structured heap and its garbage collector, two experimental systems were constructed. One creates logical areas within the heap memory of a capability computer and the other uses a simple distributed heap store. Both systems allow the execution of real programs to be measured.

5.1 Analysis of Steady State Behaviour

This analysis considers just one level of recursion in the structuring of the heap. An intuitive inductive argument could be made to assess the algorithm's performance in the general case. It is assumed that the heap is in a steady state and the division into areas is such that each behaves identically.

With the system in a steady state the rate at which garbage is generated equals the rate at which new variables are allocated. That is the total size of all the variables that become garbage per second equals the total size of all new variables allocated per second. Under these conditions the total size of all accessible variables remains constant.

Now consider the minimum size of store required to run the computation without exhausting the free store. Clearly this must be large enough to hold all the accessible variables and any inaccessible variables which have not been recovered. The maximum of this value occurs just before a garbage collection cycle is completed when the variables identified as garbage are recovered. At this point the store contains the accessible variables, the inaccessible variables that are about to be recovered and the garbage that has been generated during the cycle but which has not yet been identified as inaccessible.

The store allocated in the heap therefore gradually rises during a garbage collection cycle and abruptly falls as garbage is recovered at the end. This gives rise to the saw tooth graph shown in figure 5.1a. Note that the size of the garbage recovered is the same as the size of the garbage generated during a garbage collection cycle because the heap is assumed to be in a steady state.

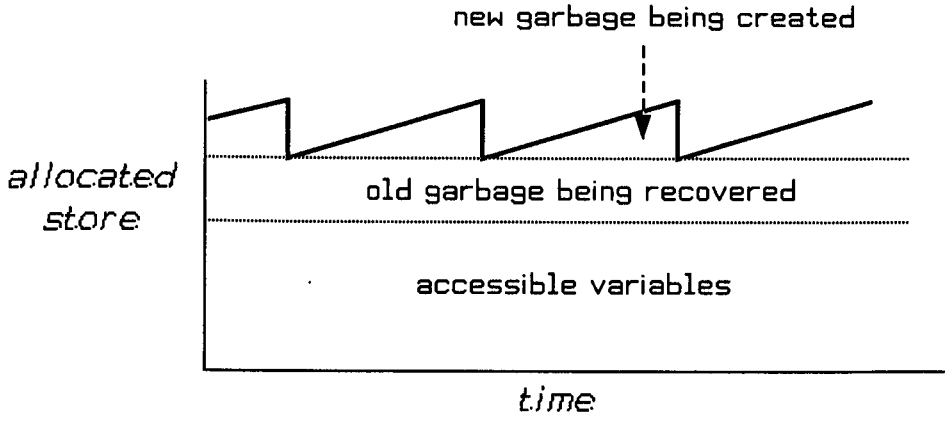


Fig5.1a Allocated store comprises accessible variables, garbage which is being recovered and garbage which is being generated.

Marking garbage collectors do not, of course, recover all the garbage instantly at the end of a garbage collection cycle. This is because they must search for those variables which are to be recovered and add them to the free list. However it will be assumed that the recovery is instant to simplify the analysis. This is a reasonable approximation given the other assumptions made about the steady state of the heap.

The effect of considering recovery time would be to slightly reduce the expression for the memory requirement. This is because some space becomes available for allocating new variables before the end of the cycle, though the cycles would take slightly longer because recovery time would have to be considered.

The heap is divided into areas and it is assumed that the time spent garbage collecting is divided equally amongst the areas. Each area is a fraction of the size of the whole heap, and therefore takes a fraction of the time to garbage collect. However it only receives a fraction of the cpu time available for garbage collection. The effect is that garbage collection of all the areas takes approximately the same time as a simple garbage collection of the whole heap.

Unfortunately, the recursively structured garbage collector causes variables that might be referenced by another area, but which are in fact inaccessible, to be scanned and kept. Not only does this decrease the memory utilisation by keeping inaccessible variables for longer than need be, it also wastes some garbage collection time by performing unnecessary scanning. This means it takes longer for the garbage collection of all the areas to complete, which in turn gives the computation more time to generate garbage.

The time required for the recursively structured garbage collector to perform a complete garbage collection of the whole heap depends on the arrangement of inter-area references. If there are none, the global garbage collection takes just one cycle of local garbage collections. However if inter-area references are present, it is not the quantity that determines how many cycles are required for a complete scan. This is actually governed by how the inter-area references thread their way through the areas, in particular a measure called remoteness.

An accessible variable's remoteness is defined as follows. This is a measure of how far a variable is from a directly accessible variable, ie. one referenced by a root. A path exists from one variable to another if it can be reached by following references, starting with one stored in the variable. The length of a path is the number of inter-area references that it follows. A variable which can be reached from a directly accessible variable on a path of length zero, has a remoteness of zero. That is all variables in an area which can be reached from the roots without following an inter-area pointer have a remoteness of zero. In general the remoteness of an accessible variable is the minimum of the lengths of all the paths to it from all directly accessible variables. The minimum value is important, rather than the maximum, because the garbage collector will find an accessible variable first by following the shortest path.

The remoteness of the entire heap is defined to be the maximum of the remoteness of all accessible variables contained in it. Figure 5.1b gives an example heap and shows the remoteness of the accessible variables.

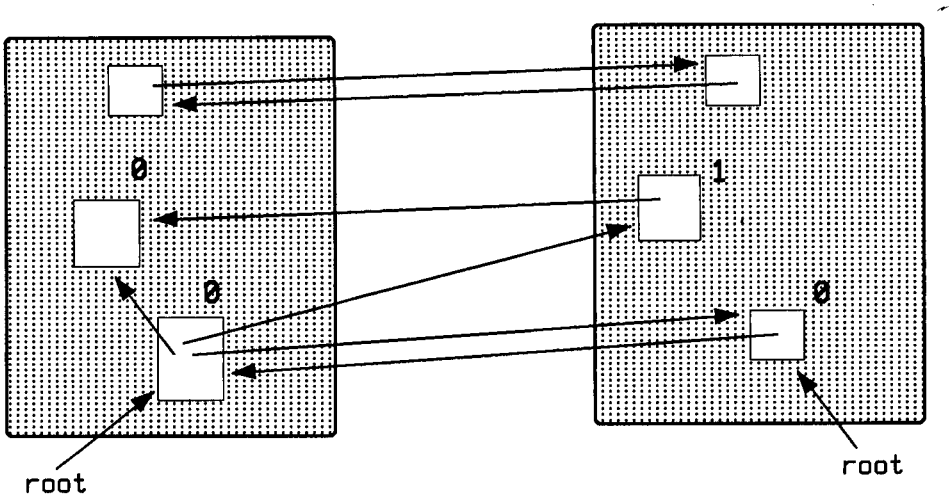


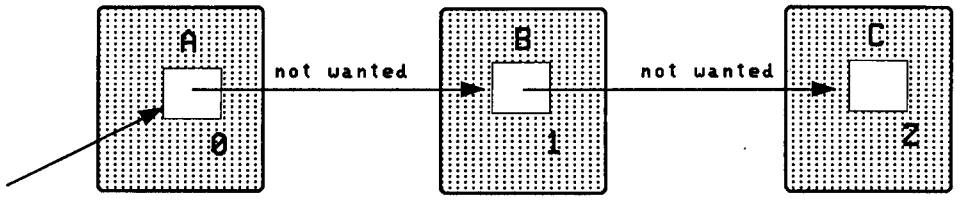
Fig5.1b An accessible variable's remoteness indicates how far it is from a root. The remoteness of the accessible variables is shown.

If a heap contains an accessible variable of remoteness **R**, at worst **R+1** local garbage collections are required to propagate the wanted mark from the roots to the variable and to scan the variable for more references. Each of the extra delays is caused by the scan in one area finding a reference on the path and marking the variable it refers to, after that variable has been scanned in the other area. The variable is scanned early because it is known that it might be accessible and all variables it refers to must be identified so they are not recovered by the local garbage collection. At the end of the local scan the variable will remain marked as wanted by the global garbage collection, so the next local garbage collection cycle will scan it and propagate the wanted mark to the next variable on the path to the remote variable.

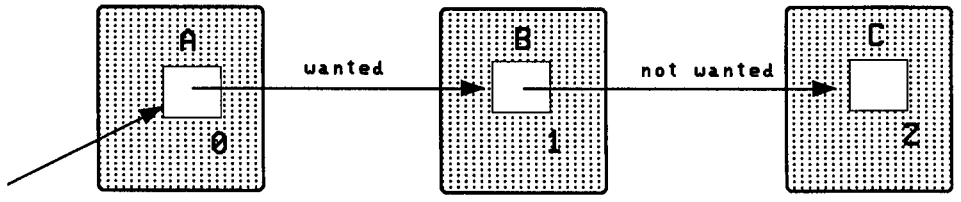
The series of diagrams in figure 5.1c show how marks are propagated along a path to a variable with a remoteness value of two. The worst case is shown which takes three local garbage collections to complete.

The worst case time for a complete global garbage collection of a heap divided into areas depends on the remoteness of the variables, not on the number of areas or number of inter-area references. This is assuming that the time taken to mark inter-area references is small compared to the time needed for a local garbage collection. The time taken to scan the whole heap is therefore that needed to complete **R+1** local garbage collections, where **R** is the remoteness of the heap.

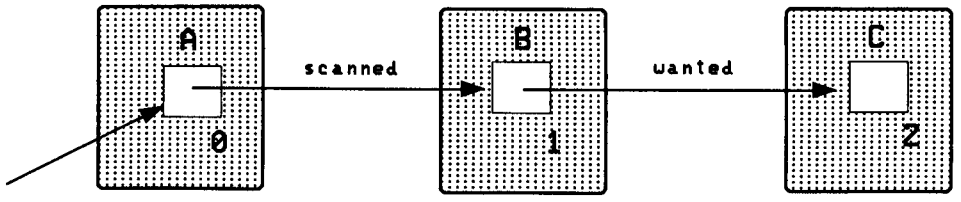
Initially inter-area references on the path are marked as not wanted.



The first cycle finds the reference from A to B and marks it wanted.



The second cycle finds the reference from B to C and marks it wanted. This means the reference from A to B has now been scanned.



The final cycle determines that C is at the end of the path, so the reference from B to C has now been scanned and the global garbage collection scan has finished.

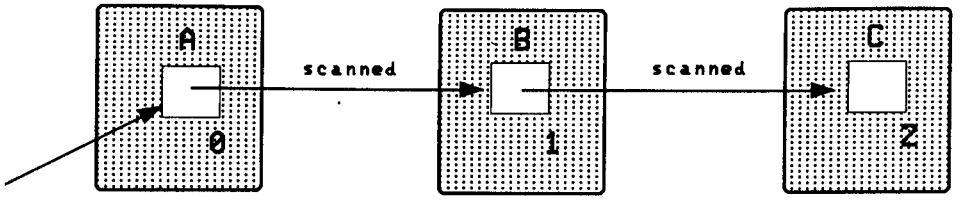


Fig5.1c Three local garbage collection cycles are required to completely scan a variable of remoteness 2.

Global garbage is not recovered immediately after the scan is completed. The recovery is performed by the next local garbage collection's recovery phase. At the end of this the amount of store allocated is at its lowest. It comprises the accessible variables, the local garbage generated during the last local garbage collection cycle, the global garbage generated during the last global cycle and the global garbage generated during the last local cycle. This is shown in figure 5.1d.

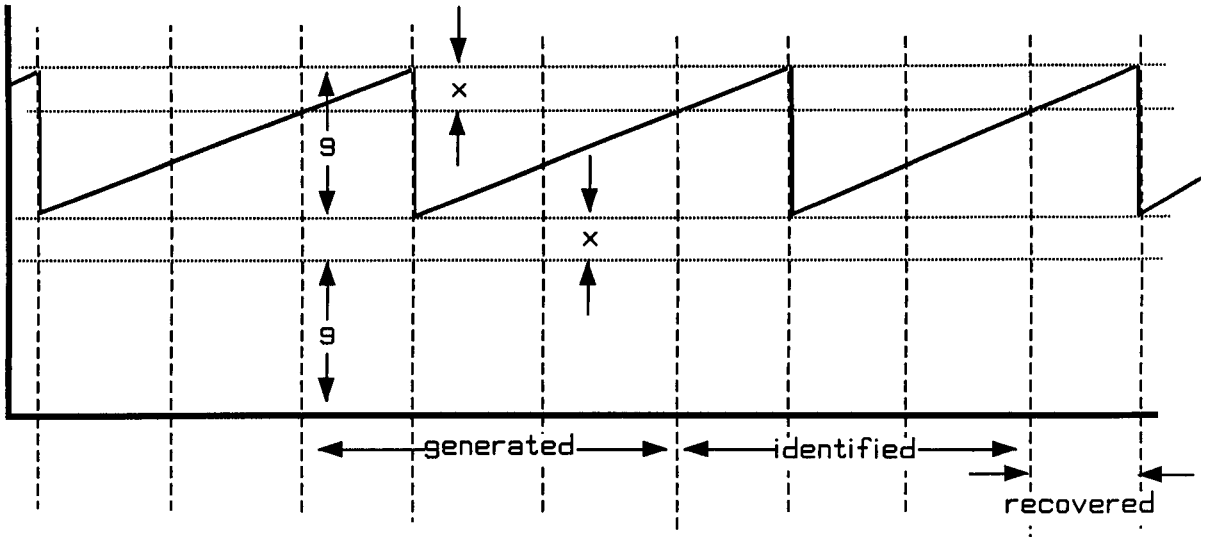


Fig5.1d Global garbage generated during R+1 local garbage collection cycles is identified during the next R+1 cycles and recovered at the end of the next local cycle.

The amount of store allocated rises during a local garbage collection cycle. It falls back when the local garbage is recovered, but not all the way. This is because some global garbage has been generated which is not recovered. Eventually the global garbage collection completes and the next local garbage collection recovers the inaccessible variables. This gives rise to the saw tooth within a saw tooth graph of figure 5.1e.

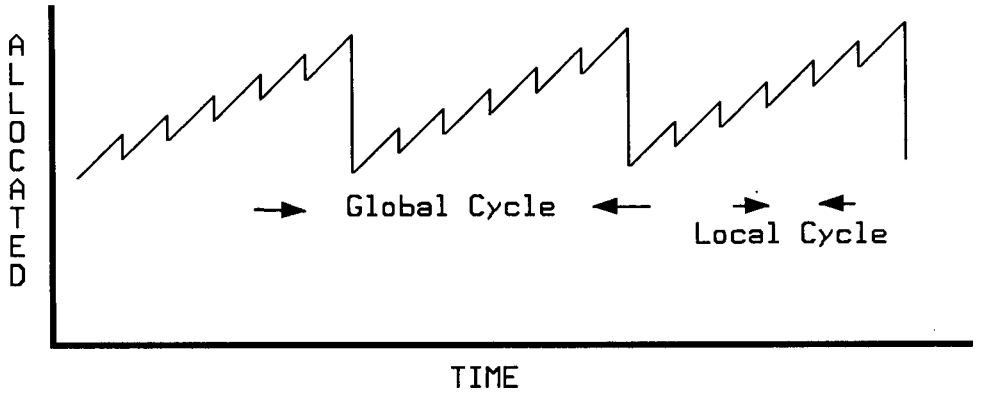


Fig5.1e When the effects of local and global garbage collections are considered together, allocated store appears like a saw tooth within a saw tooth.

The amount of global garbage generated in each global cycle depends on the time it takes to perform the local garbage collections. Successive local cycles take a little longer as more global garbage is kept, in case it later proves to be accessible, which must be scanned. To simplify the analysis, it will be assumed that the amount of global garbage is relatively small compared to local garbage and so all local cycles effectively take the same time.

When the heap is in a steady state, the garbage which is recovered by a local garbage collection and garbage which is recovered by a global garbage collection are generated at fixed rates. Their sum is equal to the rate at which new variables are created. Let r_L and r_G be the rates at which local and global garbage is generated, and s be the rate at which the accessible store is scanned. These all have units of words/time.

The maximum amount of memory allocated at any time occurs at the end of the scan of the first local cycle of a global cycle. The store contains the accessible variables, the local and global garbage generated during the local scan, the local and global garbage that is about to be recovered and the global garbage generated during the previous cycle.

$$\text{max mem} = a + 2\ t\ r_L + t\ r_G + 2\ (R+1)\ t\ r_G$$

$$= a + 2\ t\ r_L + (2R+3)\ t\ r_G$$

Each local garbage collection cycle must scan the accessible variables and any global garbage which is being kept in case it proves to be accessible. An average value for the number of scans can be used to obtain an approximate value for the time taken for each cycle.

$$s\ t = a + \frac{(R+1)\ t\ r_G}{2} \quad \Longrightarrow \quad t = \frac{2\ a}{2\ s - (R+1)\ r_G}$$

Using this expression for **t** an equation for the maximum memory requirement is obtained in terms of the remoteness of the heap, size of accessible variables, scan rate and garbage generation rates.

$$\text{max mem} = \frac{2\ s\ a + 4\ r_L\ a + 3\ R\ a\ r_G + 5\ a\ r_G}{2\ s - (R+1)\ r_G}$$

At any time only the memory containing accessible variables is usefully employed. Memory utilisation is therefore given by the ratio of the size of accessible variables to the total amount of available memory.

$$\text{util} = \frac{a}{\text{max mem}}$$

$$= \frac{2\ s - (R+1)\ r_G}{2\ s + 4\ r_L + 3\ R\ r_G + 5\ r_G}$$

In particular the utilisation expected with a heap of remoteness one is given by:

$$util_{R=1} = \frac{s - r_G}{s + 2\,r_L + 4\,r_G}$$

In section 5.3.2 this formula is used, along with results gained from measuring the garbage collection of a real distributed heap, to plot memory utilisation against the percentage of cpu time devoted to garbage collection.

5.2 Investigating Logical Areas within a Capability Computer

This experiment investigates the behaviour of the recursively structured heap using logical areas constructured within the heap memory of a capability computer. Tests were carried out using programs written for the occasion because existing applications have not been written to exploit logical areas.

5.2.1 The Experimental System

This experimental system was built into the Flex capability computer [Foster et al 82]. This is a micro coded instruction set, which runs on an ICL/Three Rivers Perq II workstation, that includes a very fast main store compacting garbage collector. References contain physical store addresses directly and variables are of varying sizes, with the first word containing the variable's size and type. References are distinguished from scalar data by tagging. Flex's garbage collector is not incremental as it uses the reference reversal technique which was described in section 2.3.2.

The experimental system assigns each variable in the heap memory to a logical area. Processes are then launched to act as the garbage collectors for each logical area and extra tables are maintained to record the state of each variable in the garbage collection. The technique described in section 4.2 was used to compress the state information.

Fortunately, four bits are spare in the word used to record the variable's size. Extra micro code was written to use these bits to record the logical area to which each variable is assigned. Logical area zero is used to indicate the roots of the heap, which for the purposes of the experiments are all the variables of the operating system and user environment.

It was not possible to modify the microcode to provide a true incremental garbage collector since memory accessing and reference manipulation occurs throughout the Flex microcode and this would effectively involve re-writing all the microcode. Instead the experimental system frequently (20ms) interrupts normal execution and inspects the state of memory to discover what has changed. Microcode assistance was provided to bypass the protection provided by the capability system and to speed up the searches. Despite this the tests run very slowly because of the complexity of the searching.

The garbage collector was written in Algol68, a source listing is provided in Appendix D. When it runs it spends some time working for each logical area. By varying these times it is possible to simulate the effects of garbage collectors working at different rates. Statistics recorded by the tests are stored on disc for later analysis.

5.2.2 Counting Words Test

In this test, an editable file is brought into main memory from backing store and is split up into words. A linked list of all the different words in the file is constructed, with a frequency count recording the number of times each word is mentioned in the text.

To compare the effectiveness of splitting the heap into logical areas, the test was run in two configurations. In the first the program used one area of the heap for all its work. In the second two areas were used, one for handling the linked list and one for all other work.

Several runs using each configuration were made using the same editable file as input. The results are plotted in figure 5.2a. It can be seen that each run for a given configuration follows the same basic pattern, though with some variation. These variations are caused by randomness in the scheduling of the garbage collector and computation processes.

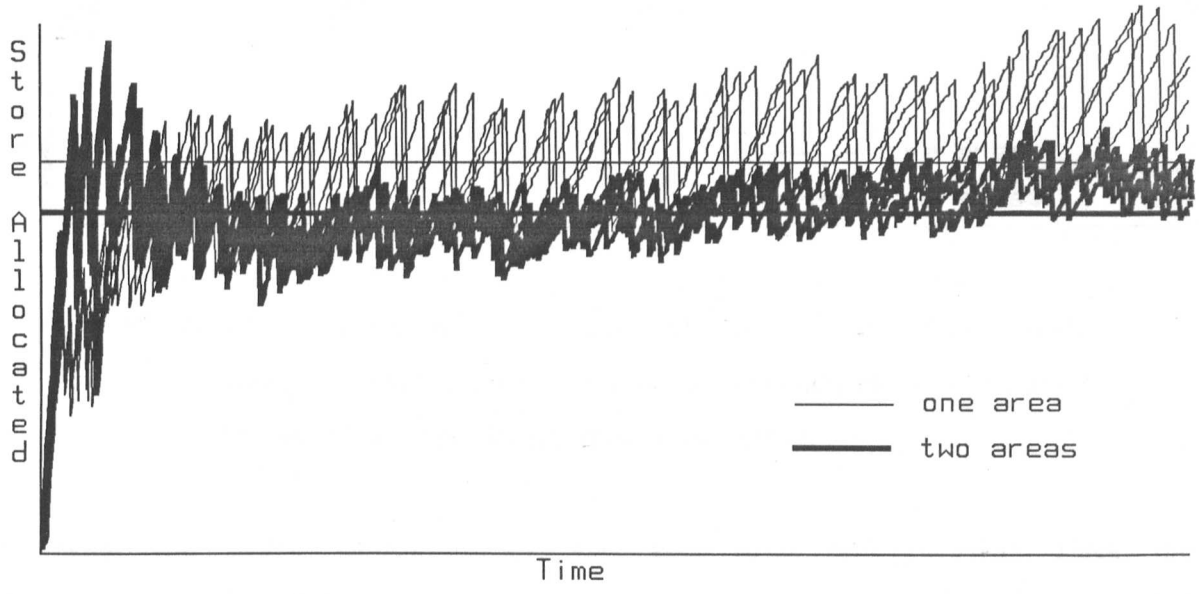


Fig5.2a The results of several runs of the count words test.

In figure 5.2b the results of only one run from each configuration are shown. Here the saw tooth shaped graph is clearly visible, and the result for two areas shows the saw tooth within a saw tooth.

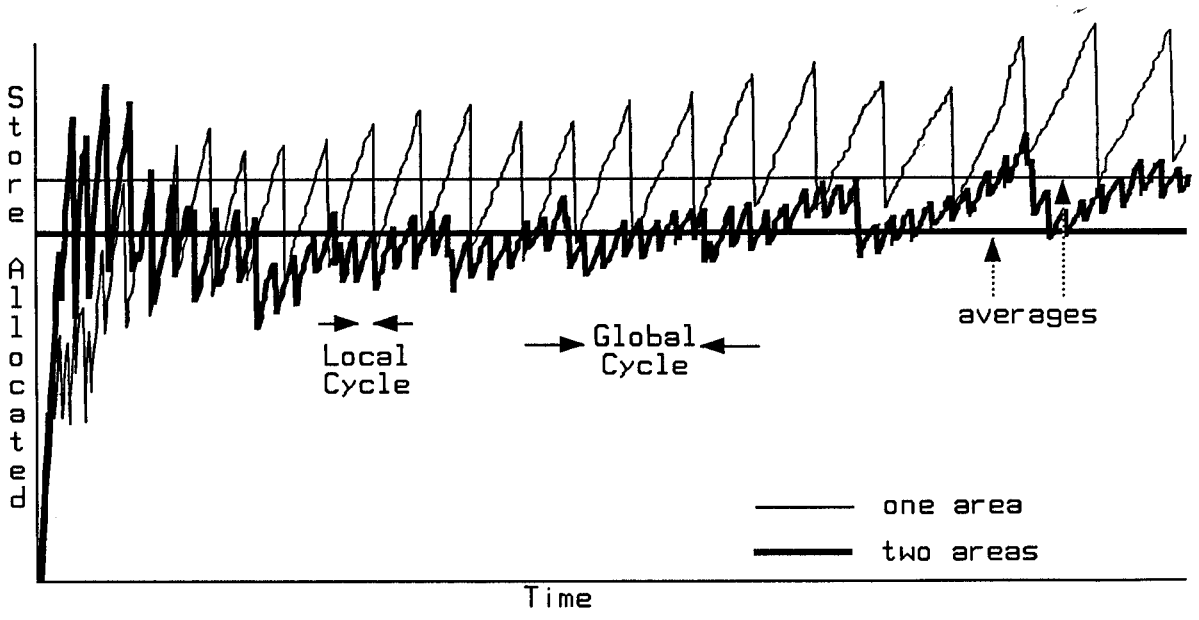


Fig5.2b The results of two runs of the count words test, showing that better memory utilisation is acheived by splitting the heap into two areas.

Initially the store allocated by the two area configuration rises more rapidly than for the single area case. This is because a relatively large number of global references are produced to start with, since many new words are encountered. These take longer to scan when using two areas and so the amount of store allocated is higher. However once a more steady state has been reached, the use of two areas clearly requires less memory. The graphs also show the average store allocated. From this it can be seen that using two areas gives a significant reduction in the amount of store required to run the programs.

5.2.3 Traffic Routing Test

In this test a simulation of traffic passing through a network is examined. Various groups of nodes of the network can be assigned to different logical areas, allowing the effects of area allocation to be studied. Also, the logical areas may be structured in a number of ways so that the effects of recursive structuring can be studied.

The first study made using this program investigates the effects of remoteness. Unfortunately it is not possible to measure this value precisely, because this would require a recursive scan of the entire heap to be performed many times. However the number of local garbage collection cycles required to complete a global garbage collection cycle can be taken as a good approximation. This is because, in the worst case, R+1 local cycles are required to perform the scan of a global area of remoteness R.

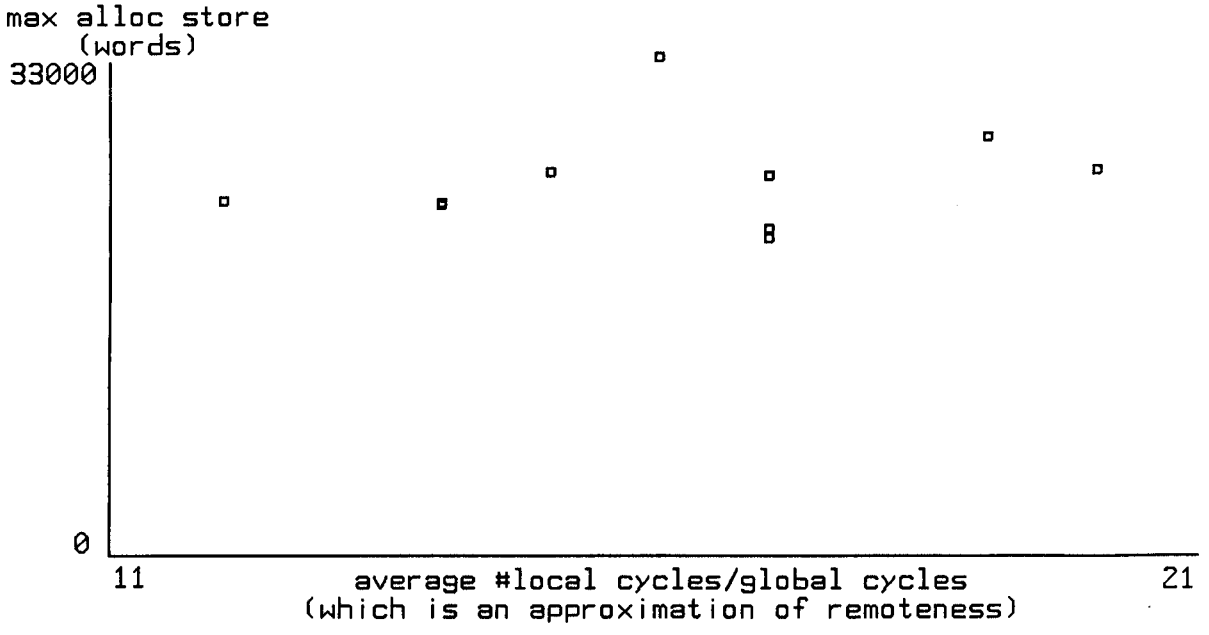


Fig5.2c: As remoteness increases, so does the amount of store required for a program to execute without interruption. However, the effect is not very significant.

The results are shown in figure 5.2c. The high average value for the number of local cycles per global is because many local garbage collections are invoked with little effect. This is because the triggering of garbage collection in the experimental system cannot be finely controlled.

The results generally confirm the prediction that, as remoteness increases so does the minimum amount of store required to execute a program. However the effect is not very pronounced. This is as predicted by the analysis of section 5.1, because the amount of global garbage produced is small compared to the amount of local garbage.

The second study shows that it can even more beneficial to structure the heap into three levels, rather than just two. The histogram in figure 5.2d show the results of using one, two or three levels with three different sets of routing data. This experiment divided the heap into three areas, but structured them into two or three levels. In the latter case, the heap was divided into two areas, one of which was divided into a further two areas, giving a total of three leaf areas.

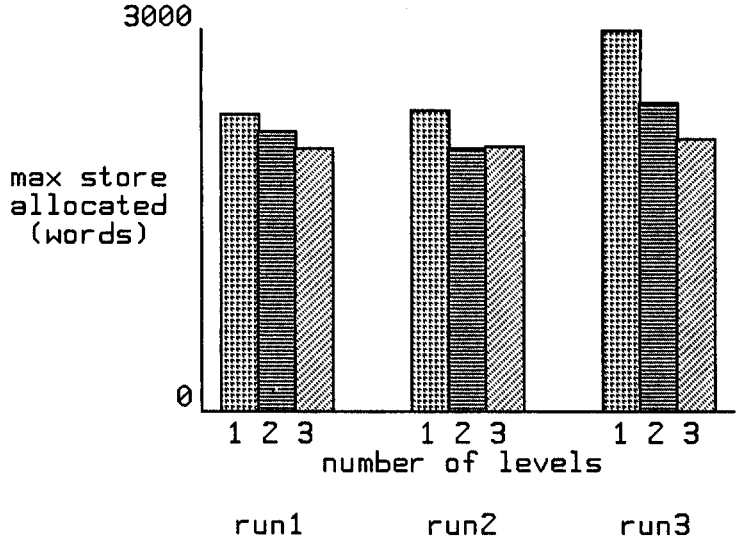


Fig5.2d: The effects of dividing the heap into more levels.

The leaf area garbage collectors were allocated different amounts of cpu time in each case, but this was adjusted by experiment to ensure that the amount of scanning performed in total was equivalent in each configuration. The histogram shows the results of three runs, using different routing data, and gives a comparison between using one level, which is effectively garbage collecting the entire heap as a whole, two levels and three levels of structuring.

The results confirm that the use of a recursively structured heap is beneficial, however they also show that it is not necessarily better to use more structuring. Dividing an area into more areas is only worthwhile if the amount of garbage which can only be recovered by global garbage collection is small compared to that which is recovered by the independent local garbage collections.

The third study using the traffic routing program investigates the benefit of allocating more CPU time to the garbage collection of areas which generate the most local garbage. Four area assignments were chosen, each structuring the heap into three logical areas arranged into two levels. For each assignment the test was run three times. In each run a different area was favoured with extra CPU time for garbage collection. The histogram in figure 5.2e shows that the store requirement for the program is least when the area the produces the most local garbage is given the most time. Conversely when the area with the least local garbage is given the most time, the store requirement is at its highest.

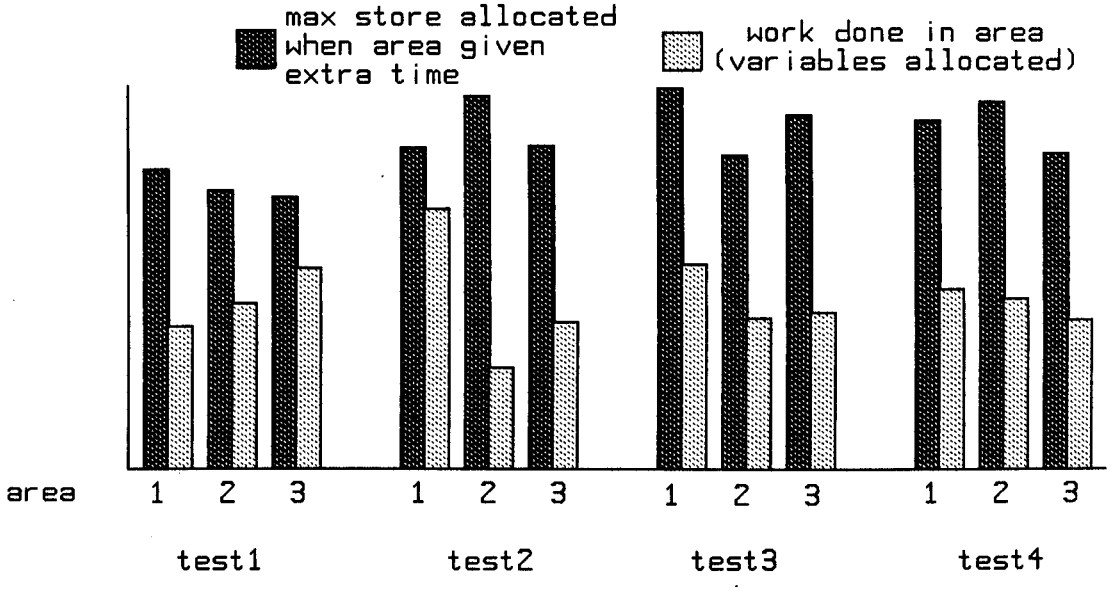


Fig5.2e: With the heap structured into two levels, the maximum store allocated is least when the area that allocates the most store is given extra CPU time for garbage collection.

More runs, using further assignments which structure the three logical areas into three levels, examine the effects of structuring the heap into more than two levels. The results, shown in figure 5.2f, are less conclusive. Only two of the results suggest that the amount of store required to execute a program is least when the area which allocates the most store is given the most CPU time for garbage collection.

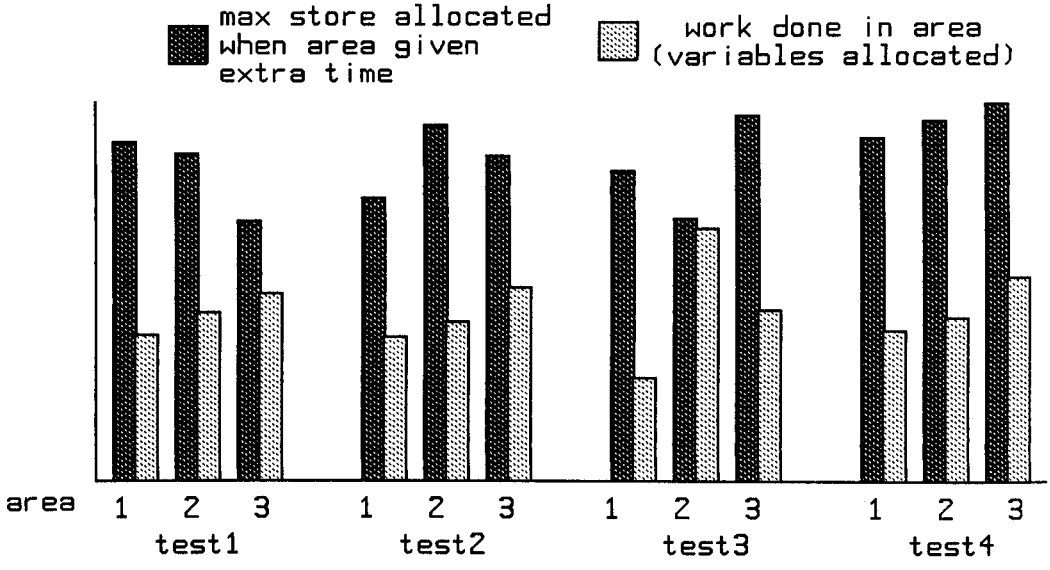


Fig5.2f: With the heap structured into three levels, it is unclear whether it is best to allocate most time to garbage collecting the area that allocates the most store.

These unexpected results, in particular the fourth test run, arise because the heap is structured asymetrically. That is the heap is divided into two areas, the second being leaf area three and the first of which is itself divided into leaf areas one and two. In these circumstances the effects of retaining inaccessible variables while global garbage collection completes, dominates the effect of recovering some garbage more quickly with the local garbage collection.

5.3 Experimenting with Garbage Collection in a Distributed Capability System

The experiments described in the previous section used specially written example programs to investigate properties of the recursively structured heap and its garbage collector. In this section results are presented which are derived from measurements taken of real applications using a real distributed heap.

5.3.1 The Experimental System

The Perq II implementation of Flex allows computers to communicate across an Ethernet using a remote procedure protocol. The remote procedures are first class objects, in that they can be passed as parameters and returned in the results of remote calls. In general, remote objects of any type can be constructed [Foster&Currie86]. The garbage collector provided for the system does not need to cater for cyclic structures, because these cannot be formed.

The second experimental system allows the relative amounts of local and global garbage, generated by programs using the distributed heap, to be measured. This was achieved by inserting statistics gathering software in the remote capabilities' garbage collector, and in the interrupt software that triggers the microcoded main store garbage collection.

Each computer holds a list of variables which are remotely referenced by other computers. The remote level garbage collector works by each computer periodically polling those computers which have referenced variables it holds. These reply indicating whether the variable is still required or not. If they are no longer required the entry is removed from the list. Inaccessible variables are recovered by a later local garbage collection. A computer removes entries from its list by using weak references, as described in section 4.8.1.

The software written for these experiments was relatively simple. The amount of store recovered by local garbage collection was recorded by modifying the interrupt software (written in Algol68) which is invoked when store is exhausted. The amount of store recovered when global garbage was recovered was measured by performing a local garbage collection before and after the inaccessible global references were destroyed.

Some of the experiments carried out involved moving editable files from one computer to another. These are structured files which can contain text and references to other editable files, though cyclic structures cannot be formed. One computer, A, can fetch an editable file from computer B. This is done by calling a remote procedure on B, supplying a procedure which stores editable files on A's disc as a remote procedure parameter. The parameter is used by B to push the requested file, and any editable files referenced by it, onto A.

The result of each push is a remote reference to the editable file on the disc of the remote computer. Thus if A fetches a highly structured file from B, many editable files will be created in A and remotely referenced by B. However, the amount of store kept accessible by the remote reference is small because the editable files are on disc and not in main memory.

5.3.2 Remote Garbage Collection Results

The most complex editable file on the Flex system contains the system's documentation. This is structured into many sub files, which explain different parts of the system. These are divided into more sub files for individual topics. Copying the documentation from one computer to another requires each sub file to be copied separately, which yields a remote reference to the sub file on the destination computer's disc. Once the reference has been incorporated into the enclosing file, the variable it refers to becomes inaccessible. The experimental system allows the amount of this globally generated garbage to be measured.

The results, given by the histograms in figure 5.3a, show that the amount of global garbage is negligible compared to the locally produced garbage. Of the garbage generated in the destination computer, only 0.6% was generated globally, while in the source this was only 0.03%. This is because the global references to the sub files only refer to variables which hold references to the files on disc, which are very small compared to the size of the files.

The source computer generated over five times more garbage than the destination computer. This is because it prepares its data for transmission by concatenating sequences, using the heap to create a new variable for the resulting sequence. The destination computer does not do this and so it uses much less store.

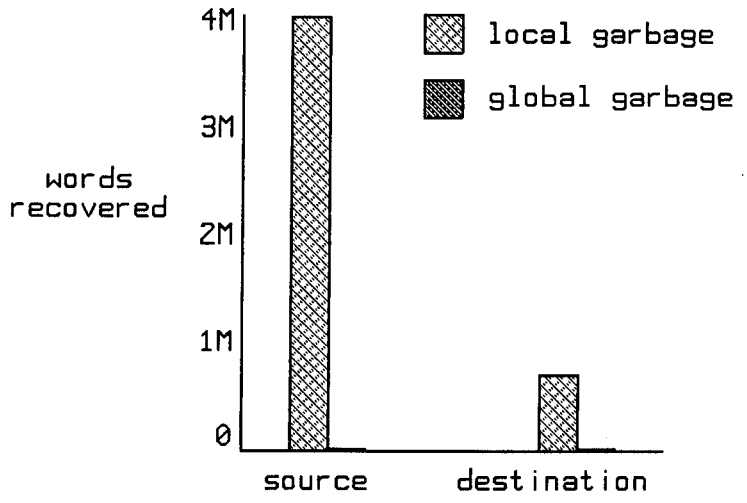


Fig5.3a: The amount of garbage produced globally is negligible compared to that produced locally.

Various programs were devised in an attempt to find an example which produced more than one percent global garbage. Unfortunately none was found. This is because inter-computer references refer either to objects on disc, which occupy very little space, or first class procedures. Procedures can keep large non-local environments accessible, but calling them creates a relatively large amount of garbage locally, which in all but the most contrived examples swamps the globally produced garbage.

Using the results obtained from these tests, some typical values for utilisation can be obtained. It was found that no more than one percent of garbage is not recovered until the completion of a global garbage collection cycle. Garbage is generated at a rate of no more than 10K words/sec. The storage which is accessible while the programs run is typically around 520K words.

Based on these results, figure 5.3b shows the expected memory utilisation plotted against the percentage of cpu time dedicated to garbage collection. This is assuming that the computation accesses memory at a rate of 5MHz.

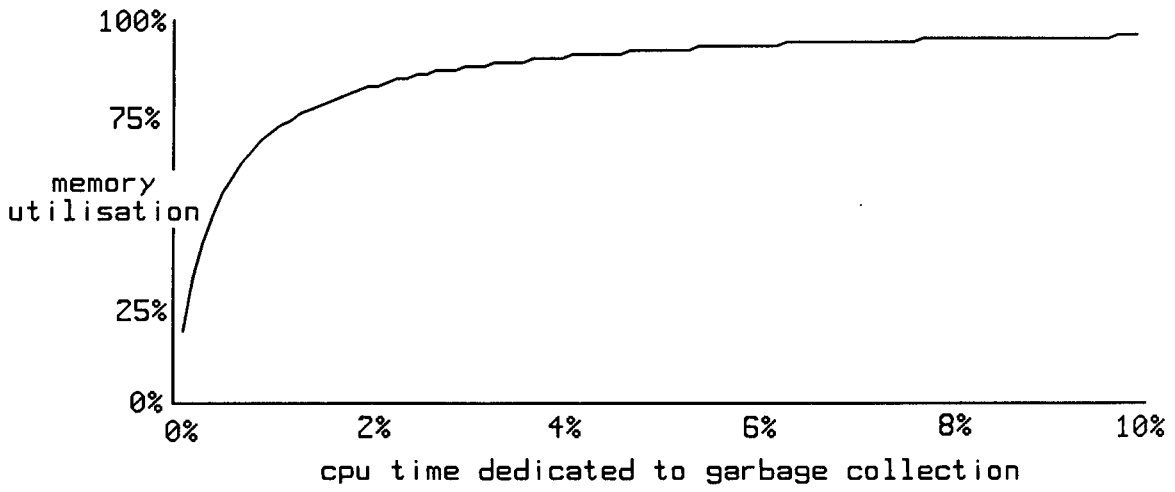


Fig5.3b: Memory utilisation predicted by experimental results

These results suggest that devoting around one percent of cpu time to garbage collection would yield a memory utilisation of about 70%. Note that other factors will, in practice, affect memory utilisation, in particular fragmentation due to allocating variables of various sizes [Randell69].

5.4 Summary

The simple analysis of the new garbage collector's performance suggests that the measure of a heap's remoteness is an important factor in determining the shape of a recursively structured heap. Minimising remoteness allows garbage collection of an internal area to proceed more quickly, which gives better memory utilisation. However, achieving this in practice may prove to be difficult but, since the amount of global garbage is likely to be negligible compared to that produced locally, this will hopefully not prove to be a serious problem.

The experiments which have been performed confirm the prediction, but show its effect to be relatively small. This is because the amount of garbage which can be recovered by local garbage collections greatly exceeds that which can only be recovered by the global garbage collector.

With independent garbage collection of separate areas of the heap, it is possible to devote more cpu time to garbage collecting some areas than to others. Experimental results show this can be beneficial. If a distributed heap is garbage collected as a whole, all computers must take part, even if they contain little garbage. By allowing independent garbage collections, those computers with less garbage are able to spend less time garbage collecting than those with more garbage. This allows each computer to utilise its cpu to the best advantage.

One problem highlighted by the experiments is that it is difficult to adjust the rate of garbage collection to the need for garbage collection. This is because the amount of garbage can only be determined by performing garbage collection. If little garbage is to be found, the cpu time spent searching for it may be better spent on the computation, whereas if too little time is spent searching the computation may exhaust the free list.

An advantage of the new algorithm is that recovering some of the inaccessible variables can be done relatively quickly. Therefore, if the free list is exhausted, the pause in execution of the computation will be shorter than if the heap were garbage collected as a whole.

6. Conclusions

This thesis has argued that it is beneficial to provide distributed systems with a heap store offering a single address space to all software in the system. A major obstacle to achieving this goal is the provision of adequate garbage collection. A survey of the literature found that no existing algorithm is entirely suitable for use in a distributed heap. Consequently a new algorithm has been developed and presented as a solution to the major problems found in other algorithms.

6.1 The Parallel Recursive Algorithm

The new garbage collection algorithm is a development of the incremental scanning garbage collector described by [Dijkstra et.al.78], [Kung&Song77] and others. The technique of divide and conquer is applied in an attempt to improve the memory utilisation of the heap. The heap is divided into a number of areas, each of which is independently garbage collected. The new algorithm shows how the effects of these independent garbage collections can be combined together to achieve garbage collection of the heap as a whole.

The new algorithm allows each area of the heap to be further divided into more areas. Thus the heap is recursively structured. The variables allocated in the heap are grouped together to form the lowest level areas. These are themselves grouped together into higher level areas, and so on until ultimately the highest level area corresponding to the whole heap is formed. The garbage collection of the areas at the lowest level of the recursively structured heap are performed in parallel. This gives the parallel recursive algorithm its name. It is important to note that the algorithm is not recursive, but the structure of the heap is.

The recursive structure imposed upon the heap serves only to control garbage collection. The heap appears as one uniform address space to its users regardless of the structure of the areas. However in a distributed system, it is likely that areas will correspond to physical resources. For example the memories of the individual computers in a distributed system may correspond to the lowest level areas. Computers grouped together on a local network would form the next level and finally groups of local networks would form the whole heap.

The garbage collecton of one area using the parallel recursive algorithm requires the use of marks on all references which pass through the area. These indicate the state of the reference in the garbage collection of the area. A reference may be 'not wanted', 'wanted', 'scanned' or 'new'. Initially all references are marked as 'not wanted'. A scan is made to find out which references are accessible. After this all references marked 'not wanted' are inaccessible and those which are accessible are marked 'scanned' or 'new'.

The garbage collector of an area does not itself perform the scan. This is done by the garbage collectors of the lower level areas each scanning their own part of the heap. The scan in made is two passes. The first starts from those references which are known to be wanted, the second starts from those references which may be wanted. Any references which pass through the area that are found to be accessible in the first pass are then known to be wanted.

Once all accessible references have been traced, those which remain marked 'not wanted' are discarded. When the next cycle of garbage collection occurs, the lower levels will not be informed of these references, so any inaccessible variables they referred to will be recovered.

6.2 Applicability

It has been shown that the parallel recursive garbage collection algorithm is applicable to a wide range of distributed system configurations. In particular it enables the individual computers in the system to perform garbage collection at their own rates and using a variety of algorithms, without significantly extra overheads.

The lowest level areas in the recursive structure may perform garbage collection using any suitable technique. In particular, if the area corresponds to the memory of a specialised processor, such as a LISP engine, it may use a specialised garbage collector which is optimised for its main use. Some changes will be required to allow the garbage collector to contribute to the higher level garbage collections, but these are relatively straightforward.

The individual systems may employ one of the incremental copying or scanning garbage collectors, a compacting garbage collector that uses pointer reversal, which is fast but causes pauses in execution, or no garbage collector at all. The latter case is suitable for systems which do not generate much garbage in the distributed heap and do not live very long. When they are switched off the variables and inter-area references they contain are immediately destroyed. Thus a distributed system can contain a mixture of time sharing systems, workstations and network servers, each with their own method of garbage collection suitable for their own needs.

The new algorithm is able to recover inaccessible cyclic structures which cross area boundaries. Here it has an advantage over methods based on reference counting which fail to do this. However it has been suggested that reference counting would usefully augment the parallel recursive algorithm, allowing non cyclic garbage to be recovered faster. The ability to recover cyclic garbage is most important for general purpose distributed systems, because otherwise erroneous or malicious software can cause large amounts of the heap to become permanently inaccessible.

To garbage collect a higher level area, the efforts of the lower level collectors are coordinated. This does not involve any computational effort or extra scanning. This coordination may be performed by the lower level garbage collector processes directly or by them communicating with an additional, special purpose process.

The different methods of coordination correspond to the different ways the parallel recursive garbage collection algorithm would be implemented in practical systems. Where no centralised control exists, such as on a local network of computers, the independent garbage collectors cooperate to control the overall garbage collection. If there is centralised control, for example when a gateway handles all traffic between two networks, the independent garbage collectors would communicate with a coordination process running in the gateway.

The algorithm is therefore applicable to a variety of physical network configurations. In addition the algorithm may be applied to individual systems, applying the recursive structuring to individual processors of a multi-processor system or to logical areas within the heap of a time sharing system. The use of logical areas allows some degree of control to be imposed upon the users of a shared heap. If each user works in a different logical area, the effects of one user on the others are minimised. In particular each user may be garbage collected independently, with cpu time for garbage collection being shared fairly.

6.3 Limitations

The new algorithm's applicability is limited by its need for extra state information to be held for inter-area references and the requirement that the lowest level garbage collectors be modified slightly.

The need to store extra state information is not a serious problem. Techniques for significantly compressing the data have been presented. With each variable that is referenced from outside its area must be stored the level of the highest area which references it, the level of the highest area in which the variable is known to be wanted and its state in that level's garbage collection. In all, this is likely to occupy less than one word.

The state information will need to be held in a table or list. This is not a significant overhead if indirection tables are necessary anyway, for example to allow the compaction of variables.

A further problem is that the algorithm requires some modification to the garbage collectors at the lowest level of the area structure. This is to arrange that they perform their tracing in two steps. While this is not difficult, it presents logistic problems when incorporating existing computers into a distributed system.

A computer's garbage collector is usually built into the lowest levels of the system, while networking software appears at a higher level. Therefore incorporating a computer into a distributed system may involve major changes, not just a modification of the networking software. In particular, if the garbage collector is microcoded or uses specialised hardware, it may be impossible to modify.

6.4 Future Research

This thesis has presented the results of research into the provision of a garbage collector for distributed systems. However a number of areas have been left largely unexplored.

In particular only a very informal attempt to show the correctness of the algorithm has been made. The complexity of the algorithm is such that its correctness is not entirely obvious. Garbage collection is performed by communicating sequential processes running in parallel and these have complex interactions.

A formal specification of the algorithm and a rigourous proof of the correctness of the refinements to implementation is an important piece of future research. A relatively large effort is required for this, due to the parallel nature of the problem.

By its nature, an incorrect garbage collector is very difficult to put right. The data structure it operates on is vast and is effectively constructed randomly by the users of the heap. A failure is likely to pass undetected for some considerable time, perhaps surfacing only when a dangling reference is followed. In such an environment it will be impossible to repeat particular failures with certainty. Testing is therefore no way to produce a correct garbage collector. Rigorous development and proof of correctness is the only way to proceed.

A deeper analysis of the new algorithm is required to assess methods for selecting the most appropriate structure for the recursive heap. Developing an appropriate statistical model is a difficult task, because the parameters involve properties of the accessible structure in the heap, such as remoteness, as well as patterns of usage like locality of reference. Better modelling would lead to a deeper understanding of the way in which the algorithm functions.

A trial implementation of the algorithm is required to show its effectiveness in practice. Comparisons would have to be made with alternative algorithms, which means a significant amount of coding is required to produce several versions of the garbage collector.

Other implementations are required to demonstrate the use of the algorithm within a computer system, working on logical areas of memory, and for garbage collecting backing store which is organised as a heap. Efficient implementation is essential for use with logical areas, because the inter-area references are likely to be manipulated far more often than inter-computer references. An implementation for garbage collecting a backing store has its own particular problems. Marking references on disc may seriously degrade disc performance and may endanger the data stored there.

One final topic for further research is an investigation of the special requirements for applicative processing on massively parallel machines. The store of such computers is organised as a heap, but the usage is very constrained. In particular, cyclic structures occur in predictable ways.

6.5 Acknowledgements

Firstly I would like to thank Antony Martin for some idle discussions about garbage collection, which eventually grew into this thesis. I am indebted to Professor Brian Randell and Dr. Derek Barnes for supervising my work and for providing tough challenges to my early unsupported claims. Thanks also go to Colin O'Halloran and Ruaridh Macdonald for their help in understanding the Z specification language and program refinement.

References

A.V.Aho, J.E.Hopcroft & J.D.Ullman
The Design and Analysis of Computer Algorithms.
Addison-Wesley series in Computer Science and Information Processing
1974

K.A.M.Ali
Object Oriented Storage Management and Garbage Collection in Distributed Processing Systems.
PhD Thesis, Royal Institute of Technology, Stockholm
December 1984

K.A.M.Ali & S.Haridi
Global Garbage Collection for Distributed Heap Storage Systems.
International Journal of Parallel Programming
Vol 15, Num 5, 1985

G.T.Almes
Garbage Collection in an Object Oriented System.
PhD Thesis
Carnegie-Mellon University, June 1980

R.K.Arora, S.P.Rana & M.N.Gupta
Ring Based Termination Detection Algorithm for Distributed Computations
Microprocessing & Microprogramming
Vol 19, June 1987, pp219..226

M.P.Atkinson & R.Morrison
Procedures as Persistent Data Objects.
ACM Trans. on Programmin Languages and Systems
Vol 7, Num 4, October 1985, pp539..559

R.C.Backhouse
Program Construction and Verification
Prentice-Hall International Series in Computer Science
1986

H.D.Baecker
Garbage Collection for Virtual Memory Computer Systems.
Comms. of the ACM
Vol 15, Num 11, November 1972, pp981..986

H.G.Baker
List Processing in Real Time on a Serial Computer.
Comms. of the ACM
Vol 21, Num 4, April 1978, pp280..294

H.G.Baker & C.Hewitt
The Incremental Garbage Collection of Processes.
ACM SIGPLAN Notices
Vol 12, Num 8, August 1977, pp55..59

B.S.Baker, E.G.Coffman & D.E.Willard
Algorithms for Resolving Conflicts in Dynamic Storage Allocation.
Journal of the ACM
Vol 32, Num 2, April 1985, pp327..343

J.M.Barth
Shifting Garbage Collection Overhead to Compile Time.
Comms. of the ACM
Vol 20, Num 7, July 1977, pp513..518

M.Ben-Ari
Algorithms for On-the-fly Garbage Collection.
ACM Trans. on Programming Languages and Systems
Vol 6, Num 3, July 1984, pp333..344

D.M.Berry & A.Sorkin
Time Required for Garbage Collection in Retention Block-Structured Languages.
Int. Jour. of Computer and Information Sciences
Vol 7, Num 4, 1978, pp361..404

K.S.Bhaskar
How Object Oriented is Your System?
ACM SIGPLAN Notices
Vol 18, Num 10, October 1983, pp8..11

A.D.Birrell & R.M.Needham
An Asynchronous Garbage Collector for the CAP Filing System.
SigOps Review
Vol 13, Num 2, April 1978, pp31..33

P.B.Bishop
Computer systems with a Very Large Address Space and Garbage Collection.
PhD Thesis
Massachusetts Institute of Technology, May 1977

D.G.Bobrow
Managing Reentrant Structures Using Reference Counts.
ACM Trans. on Programming Languages and Systems
Vol 2, Num 3, July 1980, pp269..273

M.Broy & P.Pepper
Combining Algebraic and Algorithmic Reasoning: An Approach to the Schorr-Waite Algorithm.
ACM Trans on Programming Languages and Systems
Vol 4, Num 3, July 1982, pp362..381

D.R.Brownbridge
Recursive Structure in Computer Systems.
PhD Thesis
University of Newcastle upon Tyne, July 1984

D.R.Brownbridge
Cyclic Reference Counting for Combinator Machines.
Functional Programming Languages & Computer Architectures
Springer-Verlag Lecture Notes in Computer Science
Vol 201, September 1985, pp273..288

J.A.Campbell
A Note on an Optimal-fit Method for Dynamic Allocation of Storage.
Computer Journal
Vol 14, Num 1, February 1971, pp7...9

D.J.Challab & J.D.Roberts
Buddy Algorithms
Computer Journal
Vol 30, Num 4, August 1987, pp308..315

C.J.Cheney
A Nonrecursive List Compacting Algorithm.
Comms. of the ACM
Vol 13, Num 11, November 1970, pp677..678

T.W.Christopher
Reference Count Garbage Collection.
Software - Practice and Experience
Vol 14, Num 6, June 1984, pp503..507

D.W.Clark
An Efficient List-Moving Algorithm using Constant Workspace.
Comms. of the ACM
Vol 19, Num 6, June 1976, pp352..354

D.W.Clark & C.C.Green
An Empirical Study of List Structure in LISP.
Comms. of the ACM
Vol 20, Num 2, February 1977, pp78..87

D.W.Clark & C.C.Green
A Note on Shared List Structure in LISP
Information Processing Letters
Vol 7, Num 6, October 1978, pp312..314

T.J.W.Clarke, P.J.S.Gladstone, C.D.MacLean & A.C.Norman
SKIM - The S K I Reduction Machine
Procs. Lisp 80 Conference, 1980, pp128..135

E.G.Coffman, T.T.Kadota & L.A.Shepp
On the Asymptotic Optimality of First-Fit Storage Allocation.
IEEE Trans. Software Engineering
Vol SE-11, Num 2, February 1985, pp235..239

J.Cohen
Garbage Collection of Linked Data Structures.
ACM Computing Surveys
Vol 13, Num 3, September 1981, pp341..367

J.Cohen & A.Nicolau
Comparison of Compacting Algorithms for Garbage Collection.
ACM Trans. on Programming Languages and Systems
Vol 5, Num 4, October 1983, pp532..553

G.E.Collins
A Method for Overlapping and Erasure of Lists.
Comms. of the ACM
Vol 3, 1960, pp655..657

D.Craigen
A Technical Overview of Four Verification Systems:
Gypsy, Affirm, FDM and Revised Special.
I.P.Sharp Associates, August 1985.

J.B.Dennis
On Storage Management for Advanced Programming Languages.
MIT, CSG Memo 1091
November 1974

L.P.Deutsch & D.G.Bobrow
An Efficient, Incremental, Automatic Garbage Collector.
Comms. of the ACM
Vol 19, Num 9, September 1976, pp522..526

L.P.Deutsch & A.M.Schiffman
Efficient Implementation of the Smalltalk-80 System
Procs. 11th ACM Conf. on Principles of Programming Languages
1984, pp297..302

E.W.Dijkstra
Guarded Commands, Nondeterminacy and Formal Derivation of Programs.
Comms. of the ACM
Vol 18, Num 8, pp453..457

E.W.Dijkstra, W.H.J.Feijen & A.J.M.vanGasteren
Derivation of a Termination Detection Algorithm for Distributed Computations.
Information Processing Letters
Vol 16, Num 5, June 1983, pp217..219

E.W.Dijkstra, L.Lamport, A.J.Martin, C.S.Scholten & E.F.M.Steffens
On-the-Fly Garbage Collection: An Exercise in Cooperation.
Comms. of the ACM
Vol 21, Num 11, November 1978, pp966..975

D.M.England
Capability Concept Mechanisms and Structure in System 250.
Rev. Fr. Autom. Inf. Rech. Oper. (France)
Vol 9, September 75, pp47..62

P.D.Ezhilchelvan & S.K.Shrivastava
A Characterisation of Faults in Systems
Computer Laboratory Technical Report 206
University of Newcastle upon Tyne, September 1985

R.R.Fenichel & J.C.Yochelson
A LISP Garbage-Collector for Virtual-Memory Computer Systems.
Comms. of the ACM
Vol 12, Num 11, November 1969, pp611..612

D.A.Fisher
Bounded Workspace Garbage Collection in an Address-Order Preserving List Processing Environment.
Information Processing Letters
Vol 3, Num 1, July 1974, pp29..32

J.P.Fitch & A.C.Norman
A Note on Compacting Garbage Collection.
The Computer Journal
Vol 21, Num 1, February 1978, pp31..34

J.M.Foster & I.F.Currie
Remote Capabilities in Computer Networks.
RSRE Memorandum 3947
March 1986

J.M.Foster, I.F.Currie & P.W.Edwards
Flex: A Working Computer Based on Procedure Values.
Procs. Int. Workshop on High Level Language Computer Architecture
Fort Lauderdale, Florida, December 1982
Also as: RSRE Memo 3500

J.M.Foster, C.I.Moir, I.F.Currie, J.A.McDermid, P.W.Edwards,
J.D.Morison & C.H.Pygott
An Introduction to the Flex Computer System.
RSRE Report 79016
October 1979

N.Francez
Distributed Termination.
ACM Trans. on Programming Languages & Systems
Vol 2, Num 1, pp42..55
1980

N.Francez & M.Rodeh
Achieving Distributed Termination without Freezing.
IEEE Trans. on Software Engineering
Vol SE-8, Num 3, 1982, pp287..292

D.P.Friedman & D.S.Wise
Reference Counting Can Manage the Circular Environments of
Mutual Recursion.
Information Processing Letters
Vol 8, Num 1, January 1979, pp41..45

K.E.Gorlen
An Object Oriented Class Library for C++ Programs
Software - Practice and Experience
Vol 17, Num 12, December 1987, pp899..922

D.Gries
The Science of Programming.
Spring-Verlag Texts and Monographs in Computer Science
1981

D.H.Grit & R.L.Page
Deleting Irrelevant Tasks in an Expression Oriented Multiprocessor
System.
ACM Trans on Programming Languages and Systems
Vol 3, Num 1, January 1981, pp49..59

B.K.Haddon & W.M.Waite
A Compaction Procedure for Variable-Length Storage Elements.
The Computer Journal
Vol 10, August 1967, pp162..165

R.H.Halstead
MultiLisp: A Language for Concurrent Symbolic Computation.
ACM Transactions on Programming Languages & Systems
Vol 7, Num 4, October 1985, pp501..538

W.J.Hansen
Compact List Representation: Definition, Garbage Collection, and System Implementation.
Comms. of the ACM
Vol 12, Num 9, September 1969, pp499..507

D.R.Hanson
Storage Management for an Implementation of SNOBOL4.
Software - Practice and Experience
Vol 7, 1977, pp179..192

C.L.Harrold
A Study of Store Management Policies for Incremental Garbage Collectors
RSRE Report 86018
December 1986

C.L.Harrold & S.R.Wiseman
SMITE Instruction Set Specification
RSRE Memo to appear, 1988

I.Hayes (Ed.)
Specification Case Studies.
Prentice-Hall International Series in Computer Science
1987

C.A.R.Hoare
Communicating Sequential Processes
Prentice-Hall International Series in Computer Science
1985

P.Hudak & R.M.Keller
Garbage Collection and Task Deletion in Distributed Applicative Processing Systems.
Procs. ACM Symp. LISP & Functional Programming
Carnegie-Mellon Univ., Pittsburgh. August 1982

J.Hughes
A Distributed Garbage Collection Algorithm.
Functional Programming Languages & Computer Architecture
Springer-Verlag Lecture Notes in Computer Science
Vol 201, pp256..272
September 1985

C.B.Jones
Software Development: A Rigorous Approach.
Prentice-Hall International Series in Computer Science
1980

H.B.M.Jonkers
A Fast Garbage Compaction Algorithm.
Information Processing Letters
Vol 9, Num 1, July 1979, pp26..30

A.Kaufman
Tailored-List and Recombination-Delaying Buddy Systems.
ACM Trans. on Programming Languages and Systems
Vol 6, Num 1, January 1984, pp118..125

K.C.Knowlton
A Fast Storage Allocator.
Comms. of the ACM
Vol 8, Num 10, October 1965, pp623..625

D.E.Knuth
The Art of Computer Programming, Vol 1.
Addison-Wesley, 1973

H.T.Kung & S.W.Song
An Efficient Parallel Garbage Collection System and its Correctness Proof.
Procs. 18th Ann. Symp. on Foundation of Computer Science, 1977
pp120..131

T.Kurokawa
A New Fast and Safe Marking Algorithm.
Software - Practice and Experience
Vol 11, 1981, pp671..682

L.Lamport
Garbage Collection with Multiple Processes: An Exercise in Parallelism.
Procs. 1976 Int. Conf. on Parallel Processing
pp50..54

H.Lieberman & C.Hewitt
A Real Time Garbage Collector Based on the Lifetimes of Objects.
Comms of the ACM
Vol 26, Num 6, June 1983, pp419..429

B.Liskov & R.Ladin
Highly Available Distributed Services and Fault Tolerant Distributed Garbage Collection.
MIT Computer Science Laboratory, Programming Methodology Group Memo 48
May 1986

D.B.Lomet
Scheme for Invalidating References to Freed Storage.
IBM Journal of Research and Development
January 1975

J.H.McBeth
On the Reference Counter Method.
Comms. of the ACM
Vol 6, Num 9, September 1963, p575

J.McCarthy
Recursive Functions of Symbolic Expressions and their Computation by Machine, Part 1.
Comms. of the ACM
Vol 3, 1960, pp181..195

L.Mancini & S.K.Shrivastava
Collecting Garbage while Detecting Orphans in a Distributed System is both Cheap and Efficient.
University of Newcastle upon Tyne, Computing Laboratory report SRM/452
January 1987

J.J.Martin
An Efficient Garbage Compaction Algorithm.
Comms. of the ACM
Vol 25, Num 8, August 1982, pp571..581

J.Menu, E.Sanchez & P.Sommer
HIP, A General Heap Processor.
Microprocessing and Microprogramming
Vol 20, April 1987, pp217..222

R.Milner
A Calculus of Communicating Systems.
Springer-Verlag Lecture Notes in Computer Science 92
1980

D.A.Moon
Garbage Collection in a Large LISP System.
Procs. Symp. LISP & Functional Programming
ACM, 1984, pp235..246

D.A.Moon
Architecture of the Symbolics 3600.
Procs. 12th Computer Architecture Symposium, Boston, Mass.
IEEE/ACM, June 1985, pp 76..83

I.W.Moor
An Applicative Compiler for a Parallel Machine
Procs. ACM Conf. Functional Programming
1982, pp284..293

F.L.Morris
A Time- and Space- Efficient Garbage Compaction Algorithm.
Comms. of the ACM
Vol 21, Num 8, August 1978, pp662..665

R.Morrison, A.L.Brown, R.Carrick, R.C.H.Connor, A.Dearle & M.P.Atkinson
Polymorphism, Persistence and Software Re-use in a Strongly Typed Object-oriented Environment.
Software Engineering Journal
Vol 2, Num 6, November 1987, pp199..204

K.G.Müller
On the Feasibility of Concurrent Garbage Collection.
PhD Thesis, Technische Hogeschool Delft
March 1976

R.M.Needham & R.D.H.Walker
The Cambridge CAP Computer and its Protection System.
Operating System Reviews
Vol 11, Num 5, Nov 77, pp1..10

I.A.Newman, R.P.Stallard & M.C.Woodward
Performance of Parallel Garbage Collection Algorithms.
Loughborough Univ. Computer Studies Report 166
September 1982

I.A.Newman, R.P.Stallard & M.C.Woodward
Improved Multiprocessor Garbage Collection Algorithms.
Procs. 1983 IEEE Int. Conf. on Parallel Processing
August 1983, Columbus, Ohio, pp367..368

I.A.Newman, R.P.Stallard & M.C.Woodward
A Hybrid Multiple Processor Garbage Collection Algorithm.
Computer Journal
Vol 30, Num 2, April 1987, pp119..127

I.A.Newman & M.C.Woodward
Alternative Approaches to Multiprocessor Garbage Collection.
Procs. Int. Conf. Parallel Processing
IEEE 1982

A.K.Nori
A Storage Reclamation Scheme for Applicative Multiprocessor System.
MSc Thesis, University of Utah
December 1979

I.P.Page
Analysis of a Cyclic Placement Scheme.
The Computer Journal
Vol 27, Num 1, 1984, pp18..26

I.P.Page & J.Hagins
Improving the Performance of Buddy Systems
IEEE Transactions on Computers
Vol C-35, Num 5, May 1986, pp441..447

F.Panzieri and S.K.Shrivastava
Rajdoot: A Remote Procedure Call Mechanism Supporting Orphan Detection and Killing.
University of Newcastle upon Tyne, Computing Laboratory report 200
May 1985

J.L.Peterson & T.A.Norman
Buddy Systems.
Comms. of the ACM
Vol 20, Num 4, June 1977, pp 421..431

G.Ch.Pflug
Dynamic Memory Allocation - a Markovian Analysis.
The Computer Journal
Vol 27, Num 4, November 1984, pp 328..333

F.J.Pollack, G.W.Cox, D.W.Hammerstrom, K.C.Kahn, K.K.Lai & J.R.Rattner
Supporting Ada Memory Management in the iAPX-432
SIGPLAN Notices
Vol 17, Num 4, 1982, pp117..131

P.W.Purdom & S.M.Stigler
Statistical Properties of the Buddy System.
Journal of the ACM, Vol 17, Num 4, October 1970, pp683..697

A.Ram & J.H.Patel
Parallel Garbage Collection without Synchronization Overhead.
Procs. 12th Computer Architecture Symposium, Boston, Mass.
IEEE/ACM, June 1985, pp84..90

S.P.Rana
A Distributed Solution of the Distributed Termination Problem
Information Processing Letters
Vol 17, July 1983, pp43..46

B.Randell
A Note on Storage Fragmentation and Program Segmentation.
Comms. of the ACM
Vol 12, Num 7, July 1969, pp365..372

E.M.Reingold
A Nonrecursive List Moving Algorithm.
Comms. of the ACM
Vol 16, Num 5, May 1973, pp305..307

E.Sanchez, P.Sommer, J.Menu & C.Iseli
A General Heap Processor.
IEEE Micro
Vol 7, Num 6, December 1987, pp29..40

D.W.Sandberg
Experience with an Object-Oriented Virtual Machine.
Software - Practice and Experience
Vol 18, Num 5, May 1988, pp415..425

H.Schorr & W.M.Waite
An Efficient Machine-Independent Procedure for Garbage Collection in Various List Structures.
Comms. of the ACM
Vol 10, Num 8, August 1967, pp501..506

A.Snyder
A Machine Architecture to Support an Object Oriented Language
MIT Computer Science Lab. Technical Report 209
March 1979

G.S.Sohi, E.S.Davidson & J.H.Patel
An Efficient LISP Execution Architecture with a New Representation for List Structures.
Procs. 12th Computer Architecture Symp., Boston, Mass.
IEEE/ACM, June 1985, pp 91..98

M.Stanley
Extending Data Typing Beyond the Bounds of Programming Languages.
RSRE Memo 3878
September 1985

M.Stanley
The Use of Values without Names in a Programming Support Environment.
RSRE Memo 3901
November 1985

G.L.Steele
Multiprocessing Compactifying Garbage Collection.
Comms. of the ACM
Vol 18, Num 9, September 1975, pp495..508

N.Suzuki & M.Terada
Creating Efficient Systems for Object Oriented Languages.
Procs. 11th ACM Conf. on Principles of Programming Languages
1984, pp290..296

D.C.Swinehart, P.T.Zellweger, R.J.Beach & R.B.Hagman
A Structural View of the Cedar Programming Environment.
ACM Trans. on Programming Languages and Systems
Vol 8, Num 4, October 1986, pp419..490

L.Thorelli
A Fast Compactifying Garbage Collector.
BIT
Vol 16, Num 4, 1976, pp426..441

R.W.Topor
Termination Detection for Distributed Computations.
Information Processing Letters
Vol 18, pp33..36
January 1984

D.A.Turner
A New Implementation Technique for Applicative Languages
Software - Practice and Experience
Vol 9, 1979, pp31..49

P.Tyner
iAPX-432 General Purpose Data Processor: Architecture Reference Manual
Intel Corp. January 1981

D.Ungar
Generation Scavenging: A Non-disruptive High Performance Storage Reclamation Algorithm.
ACM SigPlan Notices
Vol 19, Num 5, May 1984, pp157..167

S.C.Vestal
Garbage Collection: An Exercise in Distributed, Fault-Tolerant Programming
PhD Dissertation
Department of Computer Science, University of Washington
January 1987

P.L.Wadler
Analysis of an Algorithm for Real Time Garbage Collection.
Comms. of the ACM
Vol 19, Num 9, September 1976, pp491..500

I.Watson
An Analysis of Garbage Collection for Distributed Systems
Internal report, Flagship Project
Dept. of Computer Science, University of Manchester
August 1986

P.Watson & I.Watson
An Efficient Garbage Collection Scheme for Parallel Computer Architectures
Procs. European Conf. on Parallel Architectures and Languages
Eindhoven, The Netherlands
Springer-Verlag Lecture Notes on Computer Science, Vol 259
June 1987, pp432..443

B.Wegbreit
A Generalised Compactifying Garbage Collector.
The Computer Journal
Vol 15, Num 3, August 1972, pp204..208

J.Weizenbaum
Knotted List Structures.
Comms. of the ACM
Vol 5, Num 3, March 1962, pp161..165

J.Weizenbaum
Symmetric List Processor.
Comms. of the ACM
Vol 6, Num 9, September 1963, pp524..543

J.E.White
A High Level Framework for Network Based Resource Sharing.
Procs. National Computer Conf.
AFIPS, Vol 45, 1976, pp561..570

J.L.White
Address/Memory Management for a Gigantic LISP Environment
or, GC Considered Harmful.
Procs. LISP 80 Conference
July 1980, pp119..127

D.S.Wise
Morris's Garbage Compaction Algorithm Restores Reference Counts.
ACM Trans. on Programming Languages and Systems
Vol 1, Num 1, July 1979, pp115..120

D.S.Wise
Design for a Multiprocessing Heap with On-board Reference Counting.
Functional Programming Languages & Computer Architectures
Springer-Verlag Lecture Notes in Computer Science
Vol 201, pp289..304
September 1985

D.S.Wise & D.P.Friedman
The One-Bit Reference Count.
BIT
Vol 17, 1977, pp351..359

S.R.Wiseman
The SMITE Object Oriented Backing Store
RSRE Memorandum 4147
March 1988

S.R.Wiseman & H.S.Field-Richards
The SMITE Computer Architecture
RSRE Memorandum 4126
January 1988

W.Wulf, E.Cohen, W.Corwin, A.Jones, R.Levin, C.Pierson & F.Pollack
Hydra: The Kernel of a Multiprocessor Operating System
Comms. of the ACM
Vol 17, Num 6, June 1974, pp337..345

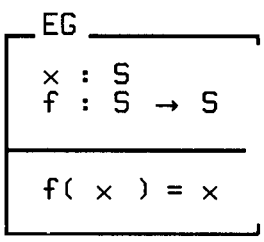
D.A.Zave
A Fast Compacting Garbage Collector
Information Processing Letters
Vol 3, Num 6, July 1975, pp167..169

Appendix A: The Z Notation

In the following , x is an identifier, T is a type, P and Q are predicates, S is a set and R is a relation.

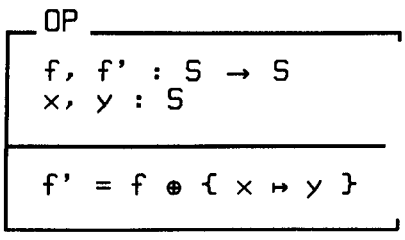
$LHS \triangleq RHS$	LHS is syntactically equivalent to RHS
$x : T$	declare x as type T
$\neg P$	not P
$P \wedge Q$	P and Q
$P \vee Q$	P or Q
$P \Rightarrow Q$	P implies Q
$\forall x : T \mid P \bullet Q$	for all x of type T, P implies Q
$\exists x : T \mid P \bullet Q$	there exists an x of type T such that P and Q
$x \in S$	x is an element of set S
$S_1 \subseteq S_2$	set S_1 is included in set S_2
$\{\}$	the empty set
$\{x_1, x_2, \ldots, x_n\}$	the set containing $x_1, x_2, \ldots, x_n$
$\mathbb{P} S$	powerset: the set of all subsets of S
$S_1 \cap S_2$	set intersection
$S_1 \cup S_2$	set union
$S_1 \setminus S_2$	set difference
$\#S$	size of finite set
$T_1 \leftrightarrow T_2$	the set of relations from T_1 to T_2
$T_1 \rightarrow T_2$	the set of total functions from T_1 to T_2
dom R	the domain of a relation R
ran R	the range of a relation R
$R_1 ; R_2$	forward relational composition
R^{-1}	inverse of a relation R
$\{a \mapsto b, c \mapsto d, \ldots\}$	the relation mapping a to b, c to d, ...
R^k	the relation R composed with itself k times
R^*	reflexive transitive closure of relation R
R^+	non-reflexive transitive closure
$R(\!(S)\!)$	relational image of set S through relation R
$S \lhd R$	domain restriction of relation R to set S
$S \lhd R$	domain subtraction
$S \rhd R$	range restriction
$S \rhd R$	range subtraction
$R_1 \oplus R_2$	overriding: $\triangleq (\text{dom } R_2 \lhd R_1) \cup R_2$

The schema notation is a way of grouping together some variable declarations and a predicate that relates them.



This schema is called EG. It declares a variable x which is drawn from the set S and a function f, from S to S. The predicate states that x and f must be such that f maps x to itself.

A schema may be included in the declarations of another, in which case the declarations of the two schemas are merged together and the predicates are conjoined.



Identifiers may be decorated. By convention a dashed variable indicates the state of a variable after an operation. Thus in the schema OP f' is the function resulting from changing f.

Appendix B: Algorithmic Notation

The algorithmic pseudocode used in Chapter 3 follows in the style of Pidgin Algol [Ahoet.al.74] and includes sets, first class procedures and parallelism in a somewhat carefree manner. This appendix describes the more obscure features by example.

refs = { r : Ref | internal(r) or incoming(r) }

refs is the set of Ref whose elememts are either internal or incoming.

FORALL r IN refs DO r.mark := not_found OD

for each member of the set refs, set its mark field to not_found.

WHILE ∃ r ∈ refs | r.mark = found DO ... OD

perform the loop while the set refs contains an element whose mark field is found

mark = λ r:Ref . IF r.mark = not_found THEN r.mark := found FI

mark is a procedure which takes a Ref and sets its mark field to found if it was not_found.

parallel(garbage_collect) (w, k, f)

create a parallel process and run the procedure garbage collect in it with parameters w, k and f.

rendezvous

wait until all processes launched so far have finished.

coroutine(a, b)

given a:(X->Void)->Void and b:(Void->X)->Void this calls procedures a and b as coroutines with procedures pa and pb as parameters. Initially b runs and a is suspended at the beginning. When b calls pb it suspends and a continues. When a calls pa with value x b continues with the value x as the result of its call of pb.

Appendix C: Proofs of Refinement

The specifications given in section 3.7.2 are repeated here for convenience. Then proofs that the propositions arising from the refinement are theorems are offered.

```
[Ref,Var]

incoming, internal: Ref ⇻ Var
outgoing : ℙ Ref
contains: Var ↔ Ref

roots: ℙ Ref
vars : ℙ Var
─────────────────────────────────────────────────────────────
outgoing ∩ dom( incoming ) = {}
outgoing ∩ dom( internal ) = {}

ran( incoming ) ∪ ran( internal ) ∪ dom( contains ) ⊆ vars
roots ∪ ran( contains ) ⊆ dom( internal ) ∪ outgoing
```

```
[X,Y]
(_ map _) : ( ℙ X × Y ) → ( X → Y )
─────────────────────────────────────────────────────────────
∀ x : ℙ X; y : Y | x ≠ {}
                • dom( x map y ) = x
                ∧ ran( x map y ) = {y}

∀ x : ℙ X; y : Y | x = {}
                • x map y = {}
```

```
[Mark]

not_found, found, scanned : Mark
─────────────────────────────────────────────────────────────
#{ not_found, found, scanned } = 3
```

```
COLLECT
─────────────────────────────────────────────────────────────
wanted, keep, accessible' : ℙ Ref
recover' : ℙ Var
─────────────────────────────────────────────────────────────
wanted ∪ keep ⊆ dom( incoming )

  accessible' = outgoing ∩ w
  recover' = vars \ (incoming ∪ internal)(| w ∪ k |)
where
  w ≙ ( internal ; contains )* (| roots ∪ wanted |)
  k ≙ ( internal ; contains )* (| keep |)
```

COLLECT1

$$keep, keep', wanted, accessible', foundin' : \mathbb{P}\ Ref$$

$$wanted \subseteq dom(\ incoming\ )$$
$$keep = keep'$$

$$accessible' = outgoing \cap w$$
$$foundin' = w \setminus outgoing$$

where

$$w \triangleq (\ internal\ ;\ contains\ )^{*}\ (\!|\ roots \cup wanted\ |\!)$$

COLLECT2

$$keep, foundin, accessible, accessible' : \mathbb{P}\ Ref$$
$$recover' : \mathbb{P}\ Var$$

$$keep \subseteq dom(\ incoming\ )$$
$$accessible' = accessible$$

$$recover' = vars \setminus (incoming \cup internal)(\!|\ foundin \cup k\ |\!)$$

where

$$k \triangleq (\ internal\ ;\ contains\ )^{*}\ (\!|\ keep\ |\!)$$

INIT1

$$marks, marks' : Ref \pfun Mark$$
$$wanted, accessible', keep, keep' : \mathbb{P}\ Ref$$

$$keep' = keep$$
$$dom(\ marks\ ) = dom(\ internal\ )$$
$$marks' = (\ dom(\ marks\ )\ map\ not\_found\ ) \oplus (\ (roots \setminus outgoing \cup wanted)\ map\ found\ )$$
$$accessible' = roots \cap outgoing$$

SCAN1

$$marks, marks' : Ref \pfun Mark$$
$$accessible, accessible', keep, keep', foundin' : \mathbb{P}\ Ref$$

$$accessible' = accessible \cup (refs \cap outgoing)$$
$$marks' = marks \oplus (refs \setminus outgoing\ map\ scanned)$$

where

$$refs \triangleq marks^{-1};(\ internal\ ;\ contains\ )^{*}\ (\!|\ \{found\}\ |\!)$$

$$keep' = keep$$

$$ran(\ marks\ ) \subseteq \{\ not\_found, found\ \}$$
$$ran(\ marks'\ ) \subseteq \{\ not\_found, scanned\ \}$$
$$dom(\ marks\ ) = dom(\ internal\ )$$
$$dom(\ marks'\ ) = dom(\ marks\ )$$

$$dom(\ marks' \rhd \{scanned\}\ ) = foundin'$$

INV

```
marks, marks' : Ref ⇸ Mark
accessible, accessible', keep, keep' : ℙ Ref
```

```
  accessible' = accessible ∪ r ∩ outgoing
where
  r ≙ marks⁻¹;(marks'⁻¹(|{scanned}|)⩤internal;contains)*(|{found}|)

marks'⁻¹(|{not_found}|) ∩ marks'⁻¹;internal;contains(|{scanned}|)={}
marks⁻¹(|{found}|) ∩ marks'⁻¹(|{not_found}|) = {}
dom( marks' ) = dom( marks )

keep' = keep
```

GUARD

```
marks : Ref ⇸ Mark
```

```
∃ next : Ref • marks( next ) = found
```

BODY

```
marks, marks' : Ref ⇸ Mark
accessible, accessible', keep, keep' : ℙ Ref
```

```
keep' = keep

∃ next : Ref | marks( next ) = found •
   marks' = marks
          ⊕ ( (marks⁻¹(|{not_found}|) ∩
               contains(|{internal(next)}|)) map found)
          ⊕ {next↦scanned}
   accessible' = accessible
                 ∪ (contains(|{internal(next)}|) ∩ outgoing)
```

```
bound : ( Ref ⇸ Mark ) → ℕ
```

```
∀ m : Ref ⇸ Mark • bound( m ) = # m ⩥ {scanned}
```

Proof of Proposition 1. pre COLLECT ⊢ pre COLLECT1

pre COLLECT ≙

```
wanted, keep : ℙ Ref
─────────────────────────────────────
wanted ∪ keep ⊆ dom( incoming )

∃ recover' : ℙ Var;
  accessible' : ℙ Ref
•
  accessible' = outgoing ∩ w
  recover' = vars \ (incoming ∪ internal)(| w ∪ k |)
where
  w ≙ ( internal ; contains )* (| roots ∪ wanted |)
  k ≙ ( internal ; contains )* (| keep |)
```

pre COLLECT1 ≙

```
wanted, keep : ℙ Ref
─────────────────────────────────────
wanted ⊆ dom( incoming )

∃ foundin' : ℙ Var;
  keep', accessible' : ℙ Ref
•
   keep' = keep

     accessible' = outgoing ∩ w
     foundin' = w \ outgoing
   where
     w ≙ ( internal ; contains )* (| roots ∪ wanted |)
```

The existential qualifier in the hypothesis is a tautology because the sets keep', recover' and accessible' can always be constructed from the given sets. Therefore the hypothesis about wanted is simply that it is a subset of dom(incoming). The conclusion therefore follows, since the existential qualifiers are always true, and hence proposition 1 is a theorem.

Proof of Proposition 2. pre COLLECT ∧ COLLECT1 ⊢ (pre COLLECT2)'

pre COLLECT ∧ COLLECT1 ≙

```
wanted, keep, keep', accessible', foundin' : ℙ Ref
─────────────────────────────────────────────────
wanted ∪ keep ⊆ dom( incoming )

∃ recover' : ℙ Var;
  accessible' : ℙ Ref
•
      accessible' = outgoing ∩ w
      recover' = vars \ (incoming ∪ internal)(| w ∪ k |)
   where
      w ≙ ( internal ; contains )* (| roots ∪ wanted |)
      k ≙ ( internal ; contains )* (| keep |)


keep' = keep

wanted ⊆ dom( incoming )

  accessible' = outgoing ∩ w
  foundin' = w \ outgoing
where
  w ≙ ( internal ; contains )* (| roots ∪ wanted |)
```

The existential quantifier can be simplified.

≙

```
wanted, keep, keep', accessible', foundin' : ℙ Ref
─────────────────────────────────────────────────
wanted ∪ keep ⊆ dom( incoming )
keep' = keep

  accessible' = outgoing ∩ w
  foundin' = w \ outgoing
where
  w ≙ ( internal ; contains )* (| roots ∪ wanted |)
```

Since the conclusion is not concerned with keep and accessible' these can be discarded.

≙

```
wanted, keep', foundin' : ℙ Ref
---
wanted ∪ keep' ⊆ dom( incoming )
foundin'= ((internal;contains)* (( rootsUwanted )) ) \ outgoing
```

(pre COLLECT2)' ≙

```
keep', foundin', accessible' : ℙ Ref
---
keep' ⊆ dom( incoming )

∃ accessible'' : ℙ Ref
  recover' : ℙ Var
•
  accessible' = accessible''
    recover' = vars \ (incoming ∪ internal)(( foundin' ∪ k ))
  where
    k ≙ ( internal ; contains )* (( keep' ))
```

There will always be a suitable set accessible'' and since we can always find a suitable set keep' there will always be corresponding sets k and recover' given found'. Hence the conclusion is always true.

Proof of Proposition 3. pre COLLECT$_\wedge$COLLECT1$_\wedge$COLLECT2' $\vdash$ COLLECT[_''/_']

pre COLLECT$_\wedge$COLLECT1$_\wedge$COLLECT2' $\triangleq$

```
wanted, keep, accessible', foundin', keep' : ℙ Ref
accessible'' : ℙ Ref
recover'' : ℙ Var
```

```
wanted ∪ keep ⊆ dom( incoming )

∃ recover' : ℙ Var;
  accessible' : ℙ Ref
•
     accessible' = outgoing ∩ w
     recover' = vars \ (incoming ∪ internal)(| w ∪ k |)
   where
     w ≙ ( internal ; contains )* (| roots ∪ wanted |)
     k ≙ ( internal ; contains )* (| keep |)

wanted ⊆ dom( incoming )
keep' ⊆ dom( incoming )

keep' = keep
accessible'' = accessible'

  accessible' = outgoing ∩ w
  foundin' = w \ outgoing
  recover'' = vars \ (incoming ∪ internal)(| foundin' ∪ k |)
where
  w ≙ ( internal ; contains )* (| roots ∪ wanted |)
  k ≙ ( internal ; contains )* (| keep' |)
```

This is simplified by discarding the quantifier and the unused variables keep' and accessible'.

≙

```
wanted, keep, foundin', accessible'' : ℙ Ref
recover'' : Var
```

```
wanted ∪ keep ⊆ dom( incoming )

  accessible'' = outgoing ∩ w
  recover'' = vars \ (incoming ∪ internal)⦇ foundin' ∪ k ⦈
  foundin' = w \ outgoing
where
  w ≙ ( internal ; contains )* ⦇ roots ∪ wanted ⦈
  k ≙ ( internal ; contains )* ⦇ keep ⦈
```

Since the domains of incoming and internal are disjoint from the set outgoing, foundin' can be replaced by w in the definition of recover'. This leads directly to the conclusion. Hence the proposition is a theorem.

COLLECT[_''/_']

```
wanted, keep, accessible'' : ℙ Ref
recover'' : ℙ Var
```

```
wanted ∪ keep ⊆ dom( incoming )

  accessible'' = outgoing ∩ w
  recover'' = vars \ (incoming ∪ internal)⦇ w ∪ k ⦈
where
  w ≙ ( internal ; contains )* ⦇ roots ∪ wanted ⦈
  k ≙ ( internal ; contains )* ⦇ keep ⦈
```

Proof of Proposition 4. pre COLLECT1 ⊢ pre INIT1

pre COLLECT1 ≙

```
wanted, keep : ℙ Ref
```

```
wanted ⊆ dom( incoming )

∃ foundin' : ℙ Var;
  keep', accessible' : ℙ Ref
•
   keep' = keep

     accessible' = outgoing ∩ w
     foundin' = w \ outgoing
   where
     w ≙ ( internal ; contains )* ⦇ roots ∪ wanted ⦈
```

pre INIT1 ≙

```
marks : Ref ⇸ Mark
wanted, keep : ℙ Ref
─────────────────────────────────────────────
dom( marks ) = dom( internal )

∃ marks' : Ref ⇸ Mark
  accessible', keep' : ℙ Ref
•
    keep' = keep
    marks' = ( dom( marks ) map not_found )
                  ⊕ ( (roots\outgoing ∪ wanted) map found )
    accessible' = roots ∩ outgoing
```

The existential quantifiers in both the hypothesis and the conclusion are both tautologies because suitable sets foundin', accessible' and keep' and the function marks' can always be found. Therefore the proposition is a theorem because marks is not constrained by the hypothesis.

Proof of Proposition 5. pre COLLECT1 ∧ INIT1 ⊢ (pre SCAN1)'

pre COLLECT1 ∧ INIT1 ≙

```
marks, marks' : Ref ⇸ Mark
wanted, accessible', keep, keep' : ℙ Ref
─────────────────────────────────────────────
wanted ⊆ dom( incoming )

∃ foundin' : ℙ Var;
  keep', accessible' : ℙ Ref
•
   keep' = keep

     accessible' = outgoing ∩ w
     foundin' = w \ outgoing
   where
     w ≙ ( internal ; contains )* (| roots ∪ wanted |)

keep' = keep
dom( marks ) = dom( internal )
marks' = ( dom( marks ) map not_found )
                ⊕ ( (roots\outgoing ∪ wanted) map found )
accessible' = roots ∩ outgoing
```

Simplify the quantifier and the redundant variable accessible'.

≙

```
marks, marks' : Ref ⇸ Mark
wanted, keep, keep' : ℙ Ref
---
wanted ⊆ dom( incoming )

keep' = keep
dom( marks ) = dom( internal )
marks' = ( dom( marks ) map not_found )
              ⊕ ( (roots\outgoing ∪ wanted) map found )
```

(pre SCAN1)' ≙

```
marks' : Ref ⇸ Mark
accessible', keep' : ℙ Ref
---
ran( marks' ) ⊆ { not_found, found }
dom( marks' ) = dom( internal )

∃ accessible'', keep'', foundin'' : ℙ Ref
  marks'' : Ref ⇸ Mark
•
     accessible'' = accessible' ∪ (refs ∩ outgoing)
     marks'' = marks' ⊕ (refs\outgoing map scanned)
   where
     refs ≙ marks''⁻¹;( internal ; contains )* (| {found} |)

   keep'' = keep'

   ran( marks'' ) ⊆ { not_found, scanned }
   dom( marks'' ) = dom( marks' )

   dom( marks'' ▷ {scanned} ) =  foundin''
```

Simplify the quantification of keep'', foundin'' and accessible'' and the unconstrained accessible'.

≙

```
marks' : Ref ⇸ Mark
keep' : ℙ Ref
```

```
ran( marks' ) ⊆ { not_found, found }
dom( marks' ) = dom( internal )

∃ marks'' : Ref ⇸ Mark
•
      marks'' = marks' ⊕ (refs\outgoing map scanned)
   where
      refs ≙ marks'⁻¹;( internal ; contains )* (( {found} ))

   ran( marks'' ) ⊆ { not_found, scanned }
   dom( marks'' ) = dom( marks' )
```

It follows from the hypothesis that the range of marks' is {not_found,found} and that the domain of marks' is the domain of internal and incoming. Therefore the domain of marks'' is equal to that of marks' and the set refs includes all those marked found. Hence the range of marks'' is {not_found,scanned} and the proposition is a theorem.

Proof of Proposition 6. pre COLLECT1 ∧ INIT1 ∧ SCAN1' ⊢ COLLECT1[_''/_']

pre COLLECT1 ∧ INIT1 ∧ SCAN1' ≙

```
marks, marks', marks'' : Ref ⇸ Mark
wanted, accessible', accessible'' : ℙ Ref
keep, keep', keep'', foundin'' : ℙ Ref
```

```
wanted ⊆ dom( incoming )

∃ foundin' : ℙ Var;
  keep', accessible' : ℙ Ref
•
   keep' = keep

      accessible' = outgoing ∩ w
      foundin' = w \ outgoing
   where
      w ≙ ( internal ; contains )* (( roots ∪ wanted ))

keep' = keep
dom( marks ) = dom( internal )
marks' = ( dom( marks ) map not_found )
                ⊕ ( (roots\outgoing ∪ wanted) map found )
accessible' = roots ∩ outgoing

  accessible'' = accessible' ∪ (refs ∩ outgoing)
  marks'' = marks' ⊕ (refs\outgoing map scanned)
where
  refs ≙ marks'^-1;( internal ; contains )* (( {found} ))

keep'' = keep'

ran( marks' ) ⊆ { not_found, found }
ran( marks'' ) ⊆ { not_found, scanned }
dom( marks' ) = dom( internal )
dom( marks'' ) = dom( marks' )

dom( marks'' ▷ {scanned} ) =  foundin''
```

Simplify the quantifier and the redundant variables accessible' and keep'.

≙

```
marks, marks', marks'' : Ref ⇸ Mark
wanted, keep, keep'', foundin'', accessible'' : ℙ Ref
```

```
wanted ⊆ dom( incoming )

keep'' = keep

dom( marks ) = dom( internal )
marks' = ( dom( marks ) map not_found )
              ⊕ ( (roots\outgoing ∪ wanted) map found )

  accessible'' = (roots ∩ outgoing)
                              ∪ (refs ∩ outgoing)
  marks'' = marks' ⊕ (refs\outgoing map scanned)
where
  refs ≙ marks'^-1;( internal ; contains )* ⦇ {found} ⦈

ran( marks' ) ⊆ { not_found, found }
ran( marks'' ) ⊆ { not_found, scanned }
dom( marks' ) = dom( internal )
dom( marks'' ) = dom( marks' )

dom( marks'' ▷ {scanned} ) =  foundin''
```

Note that marks' only maps roots\outgoingUwanted to found and so can be replaced in the definition of refs. The definition of foundin'' can be simplified because only refs\ outgoing is mapped to scanned by marks''. Also the definition of marks and marks'' can be seen to satisfy their constraints and can be discarded since they are not required.

≙

```
wanted, accessible'', foundin'', keep, keep'' : ℙ Ref
```

```
wanted ⊆ dom( incoming )

keep'' = keep

  accessible'' = (roots ∪ refs) ∩ outgoing
  foundin'' = refs\outgoing
where
  refs ≙ ( internal ; contains )* ⦇ roots\outgoing
                                        ∪ wanted  ⦈
```

Those roots which are outgoing are not included in refs. However roots is included explicitly with refs when it is used in the definition of accessible''. Therefore the definition of refs can be simplified.

≙

```
wanted, accessible'', foundin'', keep, keep'' : ℙ Ref
─────────────────────────────────────────────────────
wanted ⊆ dom( incoming )

keep'' = keep

  accessible'' = w ∩ outgoing
  foundin'' = w \ outgoing
where
  w ≙ ( internal ; contains )* (| roots ∪ wanted |)
```

COLLECT1[_''/_'] ≙

```
wanted, accessible'', foundin'', keep, keep'' : ℙ Ref
─────────────────────────────────────────────────────
wanted ⊆ dom( incoming )

keep'' = keep

  accessible'' = outgoing ∩ w
  foundin'' = w \ outgoing
where
  w ≙ ( internal ; contains )* (| roots ∪ wanted |)
```

The proposition is a theorem because the hypothesis directly gives the conclusion.

Proof of Proposition 7. pre SCAN1 ⊢ pre(INV ∧ ¬ GUARD')

pre SCAN1 ≙

```
marks : Ref ⇸ Mark
accessible, keep : ℙ Ref
─────────────────────────────────────────────
ran( marks ) ⊆ { not_found, found }
dom( marks ) = dom( internal )

∃ accessible', keep', foundin' : ℙ Ref
  marks' : Ref ⇸ Mark
•
      accessible' = accessible ∪ (refs ∩ outgoing)
      marks' = marks ⊕ (refs\outgoing map scanned)
    where
      refs ≙ marks⁻¹;( internal ; contains )* (| {found} |)

    keep' = keep

    ran( marks' ) ⊆ { not_found, scanned }
    dom( marks' ) = dom( marks )

    dom( marks' ▷ {scanned} ) =  foundin'
```

The quantifications of found', keep' and accessible' can be simplified.

≙

```
marks : Ref ⇸ Mark
accessible, keep : ℙ Ref
─────────────────────────────────────────────
ran( marks ) ⊆ { not_found, found }
dom( marks ) = dom( internal )

∃ marks' : Ref ⇸ Mark
•
      marks' = marks ⊕ (refs\outgoing map scanned)
    where
      refs ≙ marks⁻¹;( internal ; contains )* (| {found} |)

    ran( marks' ) ⊆ { not_found, scanned }
    dom( marks' ) = dom( marks )
```

pre(INV $\wedge$ $\neg$ GUARD') $\triangleq$

$$
\begin{array}{|l}
\text{marks} : \text{Ref} \pfun \text{Mark} \\
\text{accessible}, \text{keep} : \mathbb{P}\ \text{Ref} \\
\hline
\exists \\
\quad \text{marks}' : \text{Ref} \pfun \text{Mark} \\
\quad \text{accessible}', \text{keep}' : \mathbb{P}\ \text{Ref} \\
\bullet \\
\quad \text{accessible}' = \text{accessible} \cup r \cap \text{outgoing} \\
\quad \text{where} \\
\quad\quad r \triangleq \text{marks}^{-1}; (\text{marks}'^{-1}(\!|\{\text{scanned}\}|\!) \lhd \text{internal}; \text{contains})^*(\!|\{\text{found}\}|\!) \\
\quad \text{marks}'^{-1}(\!|\{\text{not\_found}\}|\!) \\
\quad\quad \cap\ \text{marks}'^{-1}; \text{internal}; \text{contains}(\!|\{\text{scanned}\}|\!) = \{\} \\
\quad \text{marks}^{-1}(\!|\{\text{found}\}|\!) \cap \text{marks}'^{-1}(\!|\{\text{not\_found}\}|\!) = \{\} \\
\quad \text{dom}(\ \text{marks}'\ ) = \text{dom}(\ \text{marks}\ ) \\
\quad \text{keep}' = \text{keep} \\
\quad \forall\ \text{next} : \text{Ref} \bullet \text{marks}'(\ \text{next}\ ) \neq \text{found}
\end{array}
$$

The quantifications of keep' and accessible' can be simplified.

$\triangleq$

$$
\begin{array}{|l}
\text{marks} : \text{Ref} \pfun \text{Mark} \\
\text{accessible}, \text{keep} : \mathbb{P}\ \text{Ref} \\
\hline
\exists \\
\quad \text{marks}' : \text{Ref} \pfun \text{Mark} \\
\bullet \\
\quad \text{marks}'^{-1}(\!|\{\text{not\_found}\}|\!) \\
\quad\quad \cap\ \text{marks}'^{-1}; \text{internal}; \text{contains}(\!|\{\text{scanned}\}|\!) = \{\} \\
\quad \text{marks}^{-1}(\!|\{\text{found}\}|\!) \cap \text{marks}'^{-1}(\!|\{\text{not\_found}\}|\!) = \{\} \\
\quad \text{dom}(\ \text{marks}'\ ) = \text{dom}(\ \text{marks}\ ) \\
\quad \forall\ \text{next} : \text{Ref} \bullet \text{marks}'(\ \text{next}\ ) \neq \text{found}
\end{array}
$$

From the hypothesis, there exists a marks' with the correct domain and range and which differs from marks only in that it maps some elements to scanned, so nothing marked found becomes not_found. Also this has the property that only those variables reachable from found variables become marked scanned. Therefore a set of marks do exist which satisfy the conclusion given the hypothesis.

Proof of Proposition 8. pre SCAN1 ∧ INV ∧ ¬ GUARD' ⊢ SCAN1

pre SCAN1 ∧ INV ∧ ¬ GUARD' ≙

```
marks, marks' : Ref ⇸ Mark
accessible, accessible', keep, keep' : ℙ Ref
```

```
ran( marks ) ⊆ { not_found, found }
dom( marks ) = dom( internal )

∃ marks'' : Ref ⇸ Mark
  accessible'', keep'', foundin'' : ℙ Ref
•
      accessible'' = accessible ∪ (refs ∩ outgoing)
      marks'' = marks ⊕ (refs\outgoing map scanned)
    where
      refs ≙ marks⁻¹;( internal ; contains )* (( {found} ))

    keep'' = keep

    ran( marks'' ) ⊆ { not_found, scanned }
    dom( marks'' ) = dom( marks )

    dom( marks' ▷ {scanned} ) =  foundin''

  accessible' = accessible ∪ r ∩ outgoing
where
  r ≙ marks⁻¹;(marks'⁻¹((|{scanned}|))◁internal;contains)*((|{found}|))

marks'⁻¹((|{not_found}|)) ∩ marks'⁻¹;internal;contains((|{scanned}|))={}
marks⁻¹((|{found}|)) ∩ marks'⁻¹((|{not_found}|)) = {}
dom( marks' ) = dom( marks )

keep' = keep

∀ next : Ref • marks'( next ) ≠ found
```

Simplify the quantification of keep'', foundin'' and accessible'' and replace the universal quantifier by an predicate on the range of marks'. The quantification of marks'' can be simplified because the overriding of marks introduces no new elements to the domain and removes found from the range, replacing it by scanned.

≙

```
marks, marks' : Ref ⇸ Mark
accessible, accessible', keep, keep' : ℙ Ref
```

```
  accessible' = accessible ∪ r ∩ outgoing
where
  r ≙ marks⁻¹;(marks'⁻¹⦇{scanned}⦈◁internal;contains)*⦇{found}⦈

marks'⁻¹⦇{not_found}⦈ ∩ marks'⁻¹;internal;contains⦇{scanned}⦈={}
marks⁻¹⦇{found}⦈ ∩ marks'⁻¹⦇{not_found}⦈ = {}

keep' = keep

ran( marks ) ⊆ { not_found, found }
ran( marks' ) ⊆ { not_found, scanned }
dom( marks ) = dom( internal )
dom( marks' ) = dom( marks )
```

The definition of r restricts the domain of internal to those variables which are finally marked scanned. That is any marked not_found are ignored. However those initially marked found must finally be scanned and any contained in variables which are finally scanned must themselves finally be scanned. Therefore the restriction of internal's domain is unncessary.

≙

```
marks, marks' : Ref ⇸ Mark
accessible, accessible', keep, keep' : ℙ Ref
```

```
  accessible' = accessible ∪ r ∩ outgoing
where
  r ≙ marks⁻¹;( internal;contains )* ⦇ {found} ⦈

marks'⁻¹⦇{not_found}⦈ ∩ marks'⁻¹;internal;contains⦇{scanned}⦈={}
marks⁻¹⦇{found}⦈ ∩ marks'⁻¹⦇{not_found}⦈ = {}

keep' = keep

ran( marks ) ⊆ { not_found, found }
ran( marks' ) ⊆ { not_found, scanned }
dom( marks ) = dom( internal )
dom( marks' ) = dom( marks )
```

In this form the hypothesis can be seen to satisfy the definition of accessible' in the conclusion and a suitable set found' can always be found.The definition of marks' satisfies the constraints of the hypothesis because its domain is that of marks and its range does not include found.

SCAN1

$$marks, marks' : Ref \pfun Mark$$
$$accessible, accessible', keep, keep', foundin' : \mathbb{P}\ Ref$$

$$accessible' = accessible \cup (refs \cap outgoing)$$
$$marks' = marks \oplus (refs \setminus outgoing \mapsto scanned)$$

where

$$refs \triangleq marks^{-1}; (\ internal\ ;\ contains\ )^{*}\ (\!|\ \{found\}\ |\!)$$

$$keep' = keep$$

$$ran(\ marks\ ) \subseteq \{\ not\_found, found\ \}$$
$$ran(\ marks'\ ) \subseteq \{\ not\_found, scanned\ \}$$
$$dom(\ marks\ ) = dom(\ internal\ )$$
$$dom(\ marks'\ ) = dom(\ marks\ )$$

$$dom(\ marks' \rhd \{scanned\}\ ) = foundin'$$

Proof of Proposition 9. pre SCAN1 $\wedge$ INV $\wedge$ GUARD' $\vdash$ (pre BODY)'

pre SCAN1 $\wedge$ INV $\wedge$ GUARD' $\triangleq$

$$
\begin{array}{l}
\mathit{marks}, \mathit{marks}' : \mathit{Ref} \pfun \mathit{Mark} \\
\mathit{accessible}, \mathit{accessible}', \mathit{keep}, \mathit{keep}' : \mathbb{P}\ \mathit{Ref} \\
\hline
\mathit{ran}(\ \mathit{marks}\ ) \subseteq \{\ \mathit{not\_found}, \mathit{found}\ \} \\
\mathit{dom}(\ \mathit{marks}\ ) = \mathit{dom}(\ \mathit{internal}\ ) \\
\\
\exists\ \mathit{marks}'' : \mathit{Ref} \pfun \mathit{Mark} \\
\quad \mathit{accessible}'', \mathit{keep}'', \mathit{foundin}'' : \mathbb{P}\ \mathit{Ref} \\
\bullet \\
\qquad \mathit{accessible}'' = \mathit{accessible} \cup (\mathit{refs} \cap \mathit{outgoing}) \\
\qquad \mathit{marks}'' = \mathit{marks} \oplus (\mathit{refs} \setminus \mathit{outgoing}\ \mathit{map}\ \mathit{scanned}) \\
\quad \mathbf{where} \\
\qquad \mathit{refs} \triangleq \mathit{marks}^{-1}; (\ \mathit{internal}\ ;\ \mathit{contains}\ )^{*}\ (\!|\ \{\mathit{found}\}\ |\!) \\
\\
\quad \mathit{keep}'' = \mathit{keep} \\
\\
\quad \mathit{ran}(\ \mathit{marks}''\ ) \subseteq \{\ \mathit{not\_found}, \mathit{scanned}\ \} \\
\quad \mathit{dom}(\ \mathit{marks}''\ ) = \mathit{dom}(\ \mathit{marks}\ ) \\
\\
\quad \mathit{dom}(\ \mathit{marks}' \rhd \{\mathit{scanned}\}\ ) = \ \mathit{foundin}'' \\
\\
\ \ \mathit{accessible}' = \mathit{accessible} \cup r \cap \mathit{outgoing} \\
\mathbf{where} \\
\ \ r \triangleq \mathit{marks}^{-1}; (\mathit{marks}'^{-1}(\!|\{\mathit{scanned}\}|\!) \lhd \mathit{internal}; \mathit{contains})^{*}(\!|\{\mathit{found}\}|\!) \\
\\
\mathit{marks}'^{-1}(\!|\{\mathit{not\_found}\}|\!) \cap \mathit{marks}'^{-1}; \mathit{internal}; \mathit{contains}(\!|\{\mathit{scanned}\}|\!) = \{\} \\
\mathit{marks}^{-1}(\!|\{\mathit{found}\}|\!) \cap \mathit{marks}'^{-1}(\!|\{\mathit{not\_found}\}|\!) = \{\} \\
\mathit{dom}(\ \mathit{marks}'\ ) = \mathit{dom}(\ \mathit{marks}\ ) \\
\\
\mathit{keep}' = \mathit{keep} \\
\\
\exists\ \mathit{next} : \mathit{Ref} \bullet \mathit{marks}'(\ \mathit{next}\ ) = \mathit{found}
\end{array}
$$

(**pre** BODY)' ≙

```
marks' : Ref ⇸ Mark
accessible', keep' : ℙ Ref
```

```
∃
    marks'' : Ref ⇸ Mark
    accessible'', keep'' : ℙ Ref
    next : Ref
|
    marks'( next ) = found
•
    keep'' = keep'

    marks'' = marks'
            ⊕ ( (marks''⁻¹(|{not_found}|) ∩
                 contains(|{internal(next)}|)) map found)
            ⊕ {next↦scanned}
    accessible'' = accessible'
                   ∪ (contains(|{internal(next)}|) ∩ outgoing)
```

A suitable keep'' can always be chosen, as can accessible'' and marks'' and it follows from the hypothesis that there exists a suitable next.

Proof of Proposition 10.

$$\text{pre } SCAN1 \wedge INV \wedge GUARD' \wedge BODY' \vdash INV[\_''/\_'] \wedge bound(marks'') < bound(marks')$$

pre $SCAN1 \wedge INV \wedge GUARD' \wedge BODY' \triangleq$

$$marks, marks', marks'' : Ref \pfun Mark$$
$$accessible, accessible', accessible'' : \mathbb{P}\ Ref$$
$$keep, keep', keep'' : \mathbb{P}\ Ref$$

$$ran(\ marks\ ) \subseteq \{\ not\_found, found\ \}$$
$$dom(\ marks\ ) = dom(\ internal\ )$$

$$\exists\ marks'' : Ref \pfun Mark \bullet$$
$$accessible'' = accessible \cup (refs \cap outgoing)$$
$$marks'' = marks \oplus (refs \setminus outgoing \mapsto scanned)$$
$$\text{where } refs \triangleq marks^{-1};(\ internal\ ;\ contains\ )^{*}\ (\!|\ \{found\}\ |\!)$$

$$keep'' = keep$$

$$ran(\ marks''\ ) \subseteq \{\ not\_found, scanned\ \}$$
$$dom(\ marks''\ ) = dom(\ marks\ )$$

$$dom(\ marks' \rhd \{scanned\}\ ) = foundin''$$

$$accessible' = accessible \cup r \cap outgoing$$
$$\text{where } r \triangleq marks^{-1};(marks'^{-1}(\!|\{scanned\}|\!) \lhd internal;contains)^{*}(\!|\{found\}|\!)$$

$$marks'^{-1}(\!|\{not\_found\}|\!) \cap marks'^{-1};internal;contains(\!|\{scanned\}|\!) = \{\}$$
$$marks^{-1}(\!|\{found\}|\!) \cap marks'^{-1}(\!|\{not\_found\}|\!) = \{\}$$
$$dom(\ marks'\ ) = dom(\ marks\ )$$

$$keep' = keep$$

$$\exists\ next : Ref \bullet marks'(\ next\ ) = found$$

$$keep'' = keep'$$

$$\exists\ next : Ref \mid marks'(\ next\ ) = found \bullet$$
$$marks'' = marks' \oplus \{next \mapsto scanned\} \oplus (\ (marks'^{-1}(\!|\{not\_found\}|\!) \cap contains(\!|\{internal(next)\}|\!))\ map\ found)$$
$$accessible'' = accessible' \cup (contains(\!|\{internal(next)\}|\!) \cap outgoing)$$

Discard keep' which is redundant and simplify the quantifications of marks'', accessible'', keep'' and foundin''.

≙

```
marks, marks', marks'' : Ref ⇸ Mark
accessible, accessible', accessible'' : ℙ Ref
keep, keep'' : ℙ Ref
```

```
ran( marks ) ⊆ { not_found, found }
dom( marks ) = dom( internal )

  accessible' = accessible ∪ r ∩ outgoing
where
  r ≙ marks⁻¹;(marks'⁻¹⦇{scanned}⦈◁internal;contains)*⦇{found}⦈

marks'⁻¹⦇{not_found}⦈ ∩ marks'⁻¹;internal;contains⦇{scanned}⦈={}
marks⁻¹⦇{found}⦈ ∩ marks'⁻¹⦇{not_found}⦈ = {}
dom( marks' ) = dom( marks )

keep'' = keep

∃ next : Ref | marks'( next ) = found •
   marks'' = marks'
           ⊕ ( (marks'⁻¹⦇{not_found}⦈ ∩
                contains⦇{internal(next)}⦈) map found)
           ⊕ {next↦scanned}
   accessible'' = accessible'
                  ∪ (contains⦇{internal(next)}⦈ ∩ outgoing)
```

INV[_''/_'] ∧ bound(marks'')<bound(marks') ≙

```
marks, marks'' : Ref ⇸ Mark
accessible, accessible'', keep, keep'' : ℙ Ref
```

```
  accessible'' = accessible ∪ r ∩ outgoing
where
  r ≙ marks⁻¹;(marks''⁻¹⦇{scanned}⦈◁internal;contains)*⦇{found}⦈

marks''⁻¹⦇{not_found}⦈ ∩ marks''⁻¹;internal;contains⦇{scanned}⦈={}
marks⁻¹⦇{found}⦈ ∩ marks''⁻¹⦇{not_found}⦈ = {}
dom( marks'' ) = dom( marks )

keep'' = keep

bound(marks'')<bound(marks')
```

From the hypothesis it can be seen that marks' maps next to found and marks'' maps next to scanned. Therefore the measure decreases. A suitable value for keep'' can always be found.

The hypothesis implies that the domain of marks'' is the same as marks', because the functional overrides do not introduce new elements. Also no elements are changed to not_found and hence nothing initially marked found becomes marked not_found.

Since the contents of next which are marked not_found become marked found, and next becomes marked as scanned, no elements are introduced to marks' which cause a scanned variable to refer to a not_found variable.

From the hypothesis accessible'' is accessible' with the addition of references contained in the variable referred to by next. Since next is finally marked as scanned these will be included by the definition given in the conclusion, hence the proposition is a theorem.

Proof of Proposition 11. pre SCAN1 ∧ INV ∧ GUARD' ⊢ bound(marks') ≥ 0

pre SCAN1 ∧ INV ∧ GUARD' ≙

```
marks, marks' : Ref ⇸ Mark
accessible, accessible', keep, keep' : ℙ Ref
```

```
ran( marks ) ⊆ { not_found, found }
dom( marks ) = dom( internal )

∃ marks'' : Ref ⇸ Mark
  accessible'', keep'', foundin'' : ℙ Ref
•
      accessible'' = accessible ∪ (refs ∩ outgoing)
      marks'' = marks ⊕ (refs\outgoing map scanned)
    where
      refs ≙ marks⁻¹;( internal ; contains )* (( {found} ))

    keep'' = keep

    ran( marks'' ) ⊆ { not_found, scanned }
    dom( marks'' ) = dom( marks )

    dom( marks' ▷ {scanned} ) =  foundin''

  accessible'' = accessible ∪ r ∩ outgoing
where
  r ≙ marks⁻¹;(marks'⁻¹((|{scanned}|))◁internal;contains)*((|{found}|))

marks'⁻¹((|{not_found}|)) ∩ marks'⁻¹;internal;contains((|{scanned}|))={}
marks⁻¹((|{found}|)) ∩ marks'⁻¹((|{not_found}|)) = {}
dom( marks' ) = dom( marks )

keep' = keep

∃ next : Ref • marks'( next ) = found
```

Since there exists a next which marks' maps to found, the cardinality of marks' with scanned removed from the range must be at least one. Hence the bound is positive.

Appendix D: Source Listings

D.1 Logical Area Experiment - Algol Source

This, and the other Algol68 programs given in this appendix, is written in the Flex dialect. This allows program modules to be imported by including capabilities for them after the program's name. These appear as boxes with names in them. The implementation is otherwise as Algol68-RS except that the scope rules are relaxed, allowing values to be passed out of scope without ill effect. Thus first-class procedures are available. Basic machine instructions are used by declaring Algol operators using BIOP with a number related to the opcode.

Note that the programs use Dijkstra's naming convention of white, grey and black for not found, found and scanned respectively.

```
g_coll:

fail :Module
endof :Module
vm_modes_m :Module
pb_to_d :Module
concat :Module
intchars :Module
maxmin_m :Module
am_i_me :Module

screen :Module
vfont_m
font_table :Module
set_m

warning_m :Module

kernel_modes :Module
kernel_ops :Module
```

```
MODE PAIR   = STRUCT( INT a, b ),
     TRIPLE = STRUCT( INT a, b, c );

OP ( INT ) INT       SELECTAREA  = BIOP 1268;   { 250₈ }
OP ( INT ) INT       CLEARAREAS  = BIOP 1269;   { 251₈ }
OP ( INT ) INT       GETAREA     = BIOP 1270;   { 252₈ }
OP ( PAIR ) INT      OFFSET      = BIOP 1277;   { 261₈ }
OP ( INT ) INT       CLEARAREA   = BIOP 1278;   { 262₈ }
OP ( INT ) INT       AFTER       = BIOP 1279;   { 263₈ }
OP ( PAIR ) PAIR     CLEAROFFSET = BIOP 1295;   { 303₈ }
{
   SELECTAREA area → 0      { change current area }
   CLEARAREAS 0 → 0         { set area number of all blocks to zero }
   GETAREA ptr → area       { fetch area of given block }
   OFFSET (ptr,int) → word  { turn an integer address into a ptr }
   CLEARAREA ptr → 0        { set area number to zero }
   AFTER ptr → ptr          { ptr to next block after ptr }
}

{ record:
   1: number of blocks, #words    - store allocated
   2: area, amount of scanning    - no more greys
   3: level                       - init done
   4: level                       - scan done
   5: level                       - recover done
   6: -                           - computation complete
   7: area, #blocks, #words       - amount recovered
  10: number of areas, map, speed, delay_time
  11: pid                         - process finished
  12: number of blocks            - amount of local recovered
  13: area, level, scanned_count  - total count stats
  14: area, num alloc blocks      - blocks still allocated at the end
}


INT scavenge = BIOP 1204;
INT get_link = BIOP 1202;

MODE RVC = REF VECTOR [] CHAR;
MODE RVI = REF VECTOR [] INT;
MODE RVB = REF VECTOR [] BOOL;
MODE MAP = REF VECTOR [] REF VECTOR [] INT;
MODE GC = PROC( MAP, RVI, INT, PROC INT ) INT;
```

```
{ The following operators are essentially machine instructions. U handles
exceptions, FAIL raises them. ISPTR distinguishes between pointers and
scalars and REFINT turns a pointer and an offset into a reference. BTYPE
and BSIZE give the size and type of a block. SETD ensures that information
about exceptions is kept for debugging. FIRM converts shaky (weak) pointers
into firm (strong) ones. ABSC gives ascii code of a character. SETSLOT sets
the process slot time. LOCK applied to a procedure ensures that when
called it executes into a privileged state and is uninterruptable. PUN
applied to ref ints splits the reference into a pointer and an offset.
SETLINK sets the procedure return link for the current process. SYS
retrieves words from the machine's system block, where pointers to the
interrupt routines are kept. BREAK takes a procedure apart, giving pointers
to its code and non local data (its highly privileged!). MAKE is the
opposite. UNPACK takes a pointer and returns the contents of the block it
refers to. RETURN exits the current procedure. }

OP ( INT ) UNION( INT, ERRORPAIR )                          U = BIOP 1004;

OP ( ERRORPAIR ) INT                                       FAIL = BIOP 1273;
OP ( INT ) BOOL                                           ISPTR = BIOP 1182;
OP ( INT, INT ) REF INT                                  REFINT = BIOP 1001;
OP ( INT ) INT                                            BTYPE = BIOP 1183;
OP ( INT ) INT                                            BSIZE = BIOP 1184;
OP ( INT ) INT                                              SETD = BIOP 1194;
OP ( INT ) INT                                              FIRM = BIOP 1181;
OP ( INT ) INT                                              ABSC = BIOP 1210;

OP ( INT ) VOID                                         SETSLOT = BIOP 1316;

OP( PROC VOID ) PROC VOID                                  LOCK = BIOP 1259;
OP( PROC(VECTOR[]CHAR)VOID ) PROC(VECTOR[]CHAR)VOID        LOCK = BIOP 1259;
OP( PROC INT ) PROC INT                                    LOCK = BIOP 1259;
OP( PROC(PROC INT)INT ) PROC(PROC INT)INT                  LOCK = BIOP 1259;
OP( PROC(PROC VOID)INT ) PROC(PROC VOID)INT                LOCK = BIOP 1259;
OP( PROC(INT)VOID ) PROC(INT)VOID                          LOCK = BIOP 1259;
OP( PROC(INT)INT ) PROC(INT)INT                            LOCK = BIOP 1259;
OP( PROC(INT,INT)VOID ) PROC(INT,INT)VOID                  LOCK = BIOP 1259;
OP( PROC(INT,INT)BOOL ) PROC(INT,INT)BOOL                  LOCK = BIOP 1259;
OP( PROC(INT,INT)INT ) PROC(INT,INT)INT                    LOCK = BIOP 1259;
OP( PROC(INT,INT,INT)VOID ) PROC(INT,INT,INT)VOID          LOCK = BIOP 1259;
OP( PROC(INT,INT,INT)INT ) PROC(INT,INT,INT)INT            LOCK = BIOP 1259;

OP( GC ) GC                                                LOCK = BIOP 1259;
OP(PROC(INT,REF VECTOR[]RVB)BOOL)PROC(INT,REF VECTOR[]RVB)BOOL
                                                           LOCK = BIOP 1259;
OP( PROC(VECTOR[]INT)VOID ) PROC(VECTOR[]INT)VOID          LOCK = BIOP 1259;
OP( PROC(INT,REF INT,INT)INT ) PROC(INT,REF INT,INT)INT
                                                           LOCK = BIOP 1259;

OP ( REF INT ) STRUCT 2 INT                                 PUN = BIOP 1001;
OP ( INT ) VOID                                          SETLINK = BIOP 1303;
OP ( INT ) REF PROC VOID                                    SYS = BIOP 1317;

OP ( PROC VOID ) PAIR                                     BREAK = BIOP 1339;
OP ( PROC VOID, PAIR ) INT                                MAKE = BIOP 11240;
OP ( INT, PAIR ) INT                                      MAKE = BIOP 11240;
OP ( PROC VOID ) PROC INT                                   PUN = BIOP 1001;
OP ( INT ) PAIR                                          UNPACK = BIOP 1154;
OP ( INT ) INT                                           RETURN = BIOP 1168;
```

```
INT non_local; {bodge to ensure compiler creates lockable code}

PROC x_gc = ( MAP map, RVI speed, INT slice_time, PROC INT prog ) INT:
BEGIN

   non_local;
   { slice time is in milliseconds }
   INT slice_slots = (slice_time+15) % 16,   { 16msec = 1/60sec }
       slice_delay = ( slice_slots * 16 - slice_time ) * 62;
                                                 { 62*16 = 992 }
   {perform health checks, computer max level}
   INT max_level := 0;
   INT n = UPB map;   { this is the number of areas }
   IF n /= UPB speed
   THEN fail( "different number of speeds to areas" ) FI;
   IF n > 15 THEN fail( "Too many areas" ) FI;
   FOR i TO n
   DO
      IF UPB map[ i ] /= n
      THEN
         fail( "map not square" )
      FI;
      FOR j TO n
      DO
         IF map[ i ][ j ] > max_level
         THEN
            max_level := map[ i ][ j ]
         FI
      OD
   OD;


   scavenge;                    {force garbage collection}
   CLEARAREAS 0;                {ensure all marks are clear}
   SELECTAREA 15;               {stuff in area 15 ignored}
   VECTOR [ 500 ] INT stats;    {array to store stats}
   stats[ 1 ] := 0;
   INT n_stats := 1;

   {{  10: number of areas, map, speed, delay_time }}
   stats[ n_stats +:= 1 ] := 2 + n * n + n + 1;
   stats[ n_stats +:= 1 ] := 10;
   stats[ n_stats +:= 1 ] := n;
   FORALL ma IN map
   DO
      FORALL m IN ma
      DO
         stats[ n_stats +:= 1 ] := m
      OD
   OD;
   FORALL s IN speed
   DO
      stats[ n_stats +:= 1 ] := s
   OD;
   stats[ n_stats +:= 1 ] := slice_time;
```

```
VFONT vf = find_font( 2, 1 );   {grab suitable font}
REF [,] BOOL scr = screen[ 18:18+22-1 , 6:2UPB screen - 3 ];


PROC x_message = ( VECTOR [] CHAR m )VOID:
BEGIN
   {primitive uninterruptable code for displaying message}
   scr SET3 scr;
   INT x := 1;
   FORALL c IN m
   DO
      INT i = ABS c - first_char OF vf + 1;
      INT from = (offsets OF vf)[ i ],
          wid  = (widths OF vf)[ i ],
          to   = from + wid - 1;
      scr[ , x:x+wid-1 ] SET (bits OF vf)[ , from : to ];
      x +:= wid
   OD
END;

PROC( VECTOR [] CHAR ) VOID message = LOCK x_message;


BOOL running := TRUE;       {flags to tell us when to stop}
INT keep_collecting := 1;

INT pid := 0;


PROC x_record = ( VECTOR [] INT d ) VOID:
BEGIN
   {place some INTs into the stats buffer}
   INT save := pid;
   pid := 0;

   REF VECTOR [] CHAR str := HEAP VECTOR [0] CHAR;

   FORALL i IN d
   DO
      str := str + intchars( i ) + " "
   OD;
   message( "Record: " + str );

   IF n_stats + UPB d + 1 > UPB stats
   THEN
      {buffer full - move it to disc}
      stats[ 1 ] := pb_to_d( stats[ : n_stats ] );
      n_stats := 1
   FI;
   stats[ n_stats +:= 1 ] := UPB d;
   stats[ n_stats + 1 : n_stats +:= UPB d ] := d;

   pid := save
END;
PROC( VECTOR [] INT ) VOID record = LOCK x_record;
```

```
{characters used for states and "colours"}
CHAR init = "I", scan = "S", recover = "R",
     white = "W", grey = "G", black = "B";

{count number of greys in each area}
HEAP VECTOR [ n ] INT greys;  { all zero }
{readiness of each area at each level}
HEAP VECTOR [ n ] REF VECTOR [] BOOL ready;
   FORALL r IN ready
   DO
      r := HEAP VECTOR [ max_level ] BOOL { all FALSE }
   OD;
{state of garbage collection of a level}
HEAP VECTOR [ max_level ] CHAR state;
      FORALL s IN state DO s := init OD;

{"indirection table" held as five separate arrays}
REF VECTOR [] INT table := HEAP VECTOR [0] INT;
REF VECTOR [] CHAR xcolour := HEAP VECTOR [0] CHAR;
REF VECTOR [] INT xlevel := HEAP VECTOR [0] INT;
REF VECTOR [] CHAR colour := HEAP VECTOR [0] CHAR;
REF VECTOR [] INT xreflevel := HEAP VECTOR [0] INT;


PROC x_find = ( INT ptr ) INT:
BEGIN
   { find ptr in table - return table index }
   non_local;
   INT ind := 0;
   IF ptr /= 0
   THEN
      BITS a = BIN ABSC ptr AND 16r7ffff;
      FOR i TO UPB table WHILE ind = 0
      DO
         IF a = (BIN ABSC table[ i ] AND 16r7ffff)
         THEN
            ind := i
         FI
      OD
   FI;
   ind
END;
OP ( INT ) INT     FIND = LOCK x_find;


PROC x_size_up = INT:
BEGIN
   { count up number of bytes allocated in table }
   INT sz := 0;
   FORALL p IN table
   DO
      sz +:= BSIZE p
   OD;
   sz % 4
END;
PROC INT size_up = LOCK x_size_up;
```

```
PROC x_ready = ( INT area, level ) BOOL:
BEGIN
   {see if all levels of an area are ready}
   BOOL ok := TRUE;
   FOR i TO UPB ready WHILE ok
   DO
      ok := map[ area ][ i ] > level OREL ready[ i ][ level ]
   OD;
   ok
END;
OP ( INT, INT )BOOL                              READY = LOCK x_ready;


PROC(INT,INT)VOID scan_done;

PROC x_scan_done = ( INT area, lower_level ) VOID:
BEGIN
   {area has finished garbage collecting at level
    so work out new state}
   BOOL ok;
   INT level = lower_level + 1;
   IF state[ level ] = init
   THEN
      {finished initialising}
      ready[ area ] [ level ] := TRUE;
      IF area READY level
      THEN
         {everyone finished initialising - change to scan}
         record(( 3, level ));
         FOR i TO UPB ready
         DO
            IF map[ area ][ i ] <= level
            THEN
               ready[ i ][ level ] := FALSE
            FI
         OD;
         state[ level ] := scan
      FI
   FI;
```

```
    IF state[ level ] = scan
    THEN
       {finished scanning}
       ok := TRUE;
       FOR i TO UPB table WHILE ok
       DO
          ok := (xcolour[ i ] /= grey) OR (xlevel[ i ] < level)
       OD;
       IF ok
       THEN
          {nothing left to scan}
          ready[ area ] [ level ] := TRUE;
          IF area READY level
          THEN
             {everyone else ready too}
             FOR i TO UPB ready
             DO
                IF map[ area ][ i ] <= level
                THEN
                   ready[ i ][ level ] := FALSE
                FI
             OD;
             record(( 4, level ));
             {change to recovery state}
             state[ level ] := recover;
             IF level < max_level
             THEN
                {report that scan finished to higher level}
                scan_done( area, level )
             FI
          FI
       FI


    ELIF state[ level ] = recover
    THEN
       {completed recovery phase}
       ready[ area ] [ level ] := TRUE;
       IF area READY level
       THEN
          {everyone else ready too}
          FOR i TO UPB ready
          DO
             IF map[ area ][ i ] <= level
             THEN
                ready[ i ][ level ] := FALSE
             FI
          OD;
          record(( 5, level ));
          IF level = max_level
          THEN
             IF NOT running
             THEN
                {test has completed but keep going a bit}
                keep_collecting := ( keep_collecting - 1 ) MAX 0
             FI
          FI;
          {change to initialisation phase}
          state[ level ] := init
       FI
    FI
END;
scan_done := LOCK x_scan_done;
```

```
PROC x_external_shade = ( INT index, ptr_level, shade_level )VOID:
BEGIN
   { shade external ptr (level ptr_level) with shade_level mark }
   non_local;
   IF shade_level >= ptr_level
   THEN
      INT xlev = xlevel[ index ];
      IF xlev < shade_level
      OREL ( xlev = shade_level ANDTH xcolour[ index ] = white )
      THEN
         xcolour[ index ] := grey;
         xlevel[ index ] := shade_level
      FI
   FI
END;
PROC ( INT, INT, INT ) VOID external_shade = LOCK x_external_shade;


PROC x_update_table = ( INT u, area ) VOID:
BEGIN
   {area has completed recovery, now compress the
     indirection table and reset colours}
   INT count := 0, sz := 0, local_count := 0;
   {table is copied and compacted into new store}
   HEAP VECTOR [ u ] INT new_table;
   HEAP VECTOR [ u ] CHAR new_colour;
   HEAP VECTOR [ u ] CHAR new_xcolour;
   HEAP VECTOR [ u ] INT new_xlevel;
   HEAP VECTOR [ u ] INT new_xreflevel;
   FOR i TO UPB table
   DO
      INT ptr = table[ i ];
      INT a = GETAREA ptr;
      IF colour[ i ] = white
      ANDTH a = area
      THEN
         {block not found - 'recover' it}
         CLEARAREA ptr;
         INT btype = BTYPE ptr;
         INT bsize = BSIZE ptr;
         IF btype /= 3 ANDTH btype /= 11
         THEN
            {clear any pointers it contains}
            FOR off TO bsize % 4
            DO
               IF ISPTR OFFSET PAIR( ptr, off )
               THEN
                  CLEAROFFSET PAIR( ptr, off )
               FI
            OD
         FI;
```

```
            IF xreflevel[ i ] = 0 THEN local_count +:= 1 FI;
            sz +:= bsize
        ELSE
            {copy entry but change it to white}
            new_table[ count +:= 1 ] := table[ i ];
            new_colour[ count ] := IF a = area
                                   THEN
                                      white
                                   ELSE
                                      colour[ i ]
                                   FI;
            new_xcolour[ count ] := xcolour[ i ];
            new_xlevel[ count ] := xlevel[ i ];
            new_xreflevel[ count ] := xreflevel[ i ]
        FI
    OD;
    INT recov = UPB table - u;
    record(( 7, area, recov, sz % 4 ));
    record(( 12, local_count ));
    {remember the copies of the table}
    table := new_table;
    colour := new_colour;
    xcolour := new_xcolour;
    xlevel := new_xlevel;
    xreflevel := new_xreflevel
END;
PROC(INT,INT)VOID update_table = LOCK x_update_table;


PROC x_expand_table = VOID:
BEGIN
    {process has suspended so scan memory for new blocks}
    non_local;
    BOOL more := TRUE;
    INT count, ptr;
    REF VECTOR [] INT new_table;
    REF VECTOR [] CHAR new_colour;
    REF VECTOR [] CHAR new_xcolour;
    REF VECTOR [] INT new_xlevel;
    REF VECTOR [] INT new_xreflevel;
```

```
WHILE more
DO
   more := FALSE;
   count := 0;
   {start at the beginning if no pointers in indirection table
    otherwise start at last pointer since they are ordered}
   ptr := IF UPB table = 0 THEN 0 ELSE table[ UPB table ] FI;
   {first just count them}
   WHILE
      ptr := AFTER ptr;
      ptr /= 0
   DO
      INT a = GETAREA ptr;
      IF a /= 0 ANDTH a < 14 THEN count +:= 1 FI
   OD;

   IF count /= 0
   THEN
      {there are some new ones so expand table}
      INT old_max = UPB table;
      INT u = UPB table + count;

      new_table := HEAP VECTOR [ u ] INT;
      new_colour := HEAP VECTOR [ u ] CHAR;
      new_xcolour := HEAP VECTOR [ u ] CHAR;
      new_xlevel := HEAP VECTOR [ u ] INT;
      new_xreflevel := HEAP VECTOR [ u ] INT;

      new_table[ : UPB table ] := table;
      new_colour[ : UPB table ] := colour;
      new_xcolour[ : UPB table ] := xcolour;
      new_xlevel[ : UPB table ] := xlevel;
```

```
            count := UPB table;
            {search again}
            ptr := IF UPB table = 0 THEN 0 ELSE table[ UPB table ] FI;
            WHILE
               ptr := AFTER ptr;
               ptr /= 0
            DO
               INT a = GETAREA ptr;
               IF a /= 0 ANDTH a < 14
               THEN
                  {ignore those in area 0 - these are roots
                                        14 - these are "dead"
                                        15 - these are tables etc}
                  count +:= 1;
                  IF count <= UPB new_table
                  THEN
                     {new blocks are black}
                     new_table[ count ] := FIRM ptr;
                     new_colour[ count ] := black;
                     new_xcolour[ count ] := white;
                     new_xlevel[ count ] := 0;
                     new_xreflevel[ count ] := 0
                  ELSE
                     {some more have arrived since we last counted}
                     more := TRUE
                  FI
               FI
            OD;
            IF count < u THEN 1%0 {serious problem if we get here} FI;
            table := new_table;
            colour := new_colour;
            xcolour := new_xcolour;
            xlevel := new_xlevel;
            xreflevel := new_xreflevel
         FI
      OD
   END;
   PROC VOID expand_table = LOCK x_expand_table;
```

```
PROC x_fix_black_block = ( INT index ) VOID:
BEGIN
   { ptr to black block - shade intra-area pointers }
   non_local;
   INT ptr = table[ index ];
   INT type = BTYPE ptr;
   IF type /= 3 ANDTH type /= 11
   THEN
      INT size = ( BSIZE ptr ) % 4;
      INT from = GETAREA ptr;
      FOR i TO size
      DO
         INT v = OFFSET PAIR( ptr, i );
         IF ISPTR v
         THEN
            INT a = GETAREA v;
            IF a = from
            THEN
               INT ind = FIND v;
               IF ind /= 0 ANDTH colour[ ind ] = white
               THEN
                  { white pointer has been stored in black block }
                  greys[ a ] +:= 1;
                  colour[ ind ] := grey
               FI
            FI
         FI
      OD
   FI
END;
PROC ( INT ) VOID fix_black_block = LOCK x_fix_black_block;
```

```
PROC x_fix_xref = ( INT index ) VOID:
BEGIN
   {check block for inter-area pointers}
   non_local;
   INT ptr = table[ index ];
   INT type = BTYPE ptr;
   IF type /= 3 ANDTH type /= 11  {ignore non-pointer blocks}
   THEN
      INT size = ( BSIZE ptr ) % 4;
      INT from = GETAREA ptr;
      FOR i TO size
      DO
         INT v = OFFSET PAIR( ptr, i );
         IF ISPTR v
         THEN
            {block contains pointer}
            INT a = GETAREA v;
            IF a /= 0 ANDTH a < 14 ANDTH a /= from
            THEN
               {pointer is between areas}
               INT ind = FIND v;
               INT level = map[ a ][ from ];
               IF ind /= 0 ANDTH level > xreflevel[ ind ]
               THEN
                  {record level of inter-area reference}
                  xreflevel[ ind ] := level
               FI
            FI
         FI
      OD
   FI
END;
PROC( INT ) VOID fix_xref = LOCK x_fix_xref;
```

```
PROC x_fix_xblack_block = ( INT index ) VOID:
BEGIN
   { ptr to externally black block - shade inter-area pointers }
   non_local;
   INT ptr = table[ index ];
   INT type = BTYPE ptr;
   IF type /= 3 ANDTH type /= 11  {ignore non-pointer blocks}
   THEN
      INT size = ( BSIZE ptr ) % 4;
      INT from = GETAREA ptr;
      FOR i TO size
      DO
         INT v = OFFSET PAIR( ptr, i );
         IF ISPTR v
         THEN
            {block contains a pointer}
            INT a = GETAREA v;
            IF a /= 0 ANDTH a < 14 ANDTH a /= from
            THEN
               INT index = FIND v;
               INT lev = map[ from ][ a ];
               IF index /= 0
               ANDTH (      xcolour[ index ] = white
                       OREL xlevel[ index ] < lev    )
               ANDTH state[ lev ] /= recover
               THEN
                  { xwhite ptr has been stored in xblack block }
                  xlevel[ index ] := lev;
                  xcolour[ index ] := grey
               FI
            FI
         FI
      OD
   FI
END;
PROC ( INT ) VOID fix_xblack_block = LOCK x_fix_xblack_block;
```

```
PROC x_fix_root_block = ( INT ptr ) VOID:
BEGIN
   { ptr to root block - shade pointers to non-root blocks }
   non_local;
   INT type = BTYPE ptr;
   IF type /= 3 ANDTH type /= 11
   THEN
      INT size = ( BSIZE ptr ) % 4;
      FOR i TO size
      DO
         INT v = OFFSET PAIR( ptr, i );
         IF ISPTR v
         THEN
            INT a = GETAREA v;
            IF a /= 0 ANDTH a < 14
            THEN
               INT index = FIND v;
               IF index /= 0 ANDTH colour[ index ] = white
               THEN
                  { white pointer has been stored in root block }
                  greys[ a ] +:= 1;
                  colour[ index ] := grey
               FI
            FI
         FI
      OD
   FI
END;
PROC ( INT ) VOID fix_root_block = LOCK x_fix_root_block;
```

```
PROC fix_marks = VOID:
BEGIN
   {process suspended - now scan memory to fix up marks}
   non_local;
   INT ptr := 0;
   INT i := 0;
   IF UPB table /= 0
   THEN
      WHILE
         ptr := AFTER ptr;
         ptr /= 0
      DO
         INT a = GETAREA ptr;
         IF a = 0 OREL a = 15
         THEN
            SKIP   {ignore tables and roots blocks}
         ELIF a = 14
         THEN
            fix_root_block( ptr ) {treat dead blocks as roots}
         ELSE
            i := (i+1) FIND ptr;
            IF i /= 0
            THEN
               fix_xref( i );
               IF colour[ i ] = black
               THEN
                  fix_black_block( i )
               FI;
               IF xcolour[ i ] = black
               THEN
                  fix_xblack_block( i )
               FI
            FI
         FI
      OD
   FI
END;


PROC x_shade_starts = ( INT area, level ) VOID:
BEGIN
   {set lowest level colour according to other colours}
   non_local;
   IF    state[ level ] = recover
   ANDTH NOT ready[ area ][ level ]
   THEN
      { Clear unwanted external marks }
      FOR i TO UPB table
      DO
         IF xreflevel[ i ] = level
         ANDTH GETAREA table[ i ] = area
         THEN
            IF xlevel[ i ] < level
            THEN
               xreflevel[ i ] := 0
            FI;
            xcolour[ i ] := white;
            xlevel[ i ] := 0
         FI
      OD
   FI;
```

```
    FOR i TO UPB table
    DO
       IF GETAREA table[ i ] = area
       THEN
          {for each pointer in this area - set the colour}
          IF xcolour[ i ] = black
          ANDTH xlevel[ i ] = level
          THEN
             IF colour[ i ] = white
             THEN
                colour[ i ] := grey;
                greys[ area ] +:= 1
             FI
          ELIF xcolour[ i ] = grey
          ANDTH xlevel[ i ] = level
          THEN
             IF colour[ i ] = white
             THEN
                colour[ i ] := grey;
                greys[ area ] +:= 1
             FI;
             xcolour[ i ] := black;
             ready[ area ] [ level ] := FALSE
          ELIF xcolour[ i ] = white
          ANDTH xreflevel[ i ] = level + 1
          THEN
             IF colour[ i ] = white
             THEN
                colour[ i ] := grey;
                greys[ area ] +:= 1
             FI
          FI
       FI
    OD
END;
PROC( INT,INT ) VOID shade_starts = LOCK x_shade_starts;
```

```
PROC x_recov_area = ( INT area ) VOID:
BEGIN
   {recover blocks which are still white}
   non_local;
   INT count := 0;
   FOR i TO UPB table
   DO
      IF    colour[ i ] = white
      ANDTH GETAREA table[ i ] = area
      THEN
         count +:= 1
      FI
   OD;
   IF count /= 0
   THEN
      update_table( UPB table - count, area )
   ELSE
      { none recovered - just reset marks }
      FOR i TO UPB table
      DO
         IF GETAREA table[ i ] = area
         THEN
            IF colour[ i ] /= black THEN 1%0 FI;
            colour[ i ] := white
         FI
      OD
   FI;
   scavenge; SELECTAREA 15
END;
PROC( INT ) VOID recov_area = LOCK x_recov_area;
```

```
PROC x_init_area = ( INT area ) VOID:
BEGIN
   {initialisation phase}
   non_local;
   INT ptr := 0;
   WHILE
      ptr := AFTER ptr;
      ptr /= 0
   DO
      INT ptr_a = GETAREA ptr;
      IF ptr_a = 0
      THEN
         {found a root pointer}
         INT type = BTYPE ptr;
         IF type /= 3 ANDTH type /= 11
         THEN
            INT size = ( BSIZE ptr ) % 4;
            {search block for pointers}
            FOR i TO size
            DO
               INT v = OFFSET PAIR( ptr, i );
               IF ISPTR v
               THEN
                  INT va = GETAREA v;
                  IF va = area
                  THEN
                     {root contains pointer to block in my area}
                     INT index = FIND v;
                     IF index /= 0 ANDTH colour[ index ] = white
                     THEN
                        { white ptr has been stored in root block }
                        greys[ va ] +:= 1;
                        colour[ index ] := grey
                     FI
                  FI
               FI
            OD
         FI
      FI
   OD
END;
PROC(INT)VOID init_area = LOCK x_init_area;

PROC VOID release_processor, old_timer;
```

```
PROC x_scan_grey_block = ( INT area, level, ptr ) INT:
BEGIN
   { Shade ptrs found in block }
   non_local;
   INT type = BTYPE ptr;
   IF type /= 3 ANDTH type /= 11
   THEN
      {search block for pointers}
      INT size = ( BSIZE ptr ) % 4;
      FOR i TO size
      DO
         INT v = OFFSET PAIR( ptr, i );
         IF ISPTR v
         THEN
            INT a = GETAREA v;
            IF a /= 0 ANDTH a < 14
            THEN
               {found pointer}
               INT index = FIND v;
               IF index /= 0
               THEN
                  IF a = area
                  THEN
                     { Ptr to Same Area }
                     IF colour[ index ] = white
                     THEN
                        colour[ index ] := grey;
                        greys[ area ] +:= 1
                     FI
                  ELSE
                     INT ptr_level = map[ area ][ a ];
                     external_shade( index, ptr_level, level )
                  FI
               FI
            FI
         FI
      OD;
      size
   ELSE
      0
   FI
END;
PROC( INT,INT,INT ) INT  scan_grey_block = LOCK x_scan_grey_block;
```

```
PROC x_scan_greys = ( INT level, REF INT count, INT area ) INT:
BEGIN
   { Scan grey blocks }
   non_local;
   INT num := 0, next := 1, loops := 0;
   { scan from next, no more than count looks }
   WHILE greys[ area ] > 0
   DO
      loops +:= 1;
      IF next > UPB colour THEN next := 1 FI;
      IF colour[ next ] = grey
      ANDTH ( GETAREA table[ next ] ) = area
      THEN
         { Scan Grey Block }
         count -:= scan_grey_block( area, level, table[ next ] );
         loops := 0;
         num +:= 1;
         colour[ next ] := black;
         greys[ area ] -:= 1
      FI;
      IF count <= 0
      THEN
         {done enough for now - so relinquish processor}
         release_processor;
         loops := 0;
         count := speed[ area ]
      FI;
      IF loops > UPB table THEN 1%0 FI;
      next +:= 1
   OD;
   num  { number of grey blocks scanned }
END;
PROC(INT,REF INT,INT)INT scan_greys = LOCK x_scan_greys;
```

```
PROC x_scav = ( INT area ) INT:
BEGIN
   {the garbage collector for an area}
   non_local;
   INT count := speed[ area ];
   INT scans;
   VECTOR [ max_level ] INT scan_counts;
   FORALL s IN scan_counts DO s := 0 OD;
   BOOL collecting := TRUE;
   WHILE collecting
   DO
      {keep on going}
      collecting := keep_collecting /= 0;
      IF UPB table /= 0 THEN init_area( area ) FI;
      scans := 0;
      {perform each level of scan in turn}
      FOR level FROM max_level BY -1 TO 1
      DO
         shade_starts( area, level );
         INT s = scan_greys( level, count, area );
         scan_counts[ level ] +:= s;
         scans +:= s
      OD;
      record(( 2, area, scans ));
      IF max_level > 1
      THEN
         scan_done( area, 1 )
      ELSE
         {single level system - check we're still going}
         IF NOT running
         THEN
            keep_collecting := ( keep_collecting - 1 ) MAX 0
         FI
      FI;
      recov_area( area );
      release_processor
   OD;


   FOR l TO max_level
   DO
      record(( 13, area, l, scan_counts[ l ] ))
   OD;
   0
END;


PROC(INT)INT scav = LOCK x_scav;
ERRORPAIR failure;
BOOL failed := FALSE;

{here is modified versions of the scheduler}

SELECTAREA 14;
HEAP VECTOR [ n+1 ] INT link;
SELECTAREA 15;

INT cont_link := 0;
INT num_dead := 0;

{ pid 1 is prog, pids 2..n+1 are gcs }

INT num_slots := 0;
```

```
BOOL do_fixup := TRUE;

PROC x_run_next = VOID:
BEGIN
   INT p := pid, skips := 0;
   pid := 0;
   SETLINK cont_link;  { in case of break in }
   WHILE
      IF ( p +:= 1 ) > UPB link THEN p := 1 FI;
      IF p = 2
      THEN
         IF do_fixup
         THEN
            expand_table;
            scavenge; SELECTAREA 15;
            fix_marks
         FI;
         do_fixup := running;
         record(( 1, UPB table, size_up ))
      FI;
      link[ p ] = 0
   DO
      IF (skips +:= 1) > UPB link
      THEN
         fail( "looping" )
      FI
   OD;
   skips := 0;


   "xxx" + "yyy";   { interruptable operation }
   IF p = 1
   THEN
      SETSLOT slice_slots;            { 1 ⇒ 16 msec }
      num_slots +:= 1;
      TO slice_delay DO SKIP OD
   ELSE
      SETSLOT 200
   FI;

   INT l = link[ pid := p ];
   link[ pid ] := 0;
   SETLINK l
END;
PROC VOID run_next = LOCK x_run_next;


PROC x_release_processor = VOID:
BEGIN
   "xxx" + "yyy";   { interruptable }
   link[ pid ] := get_link;
   run_next
END;
release_processor := LOCK x_release_processor;
```

```
old_timer := (SYS 17);
PAIR old = BREAK old_timer;

PAIR ref_cp = UNPACK b OF old;   { non locals of timer is ref cp }
INT pslot = 20;

PROC x_return = ( INT ret ) VOID:
BEGIN
   non_local;
   INT link = get_link;
   SETLINK ret;
   RETURN link
END;
PROC ( INT ) VOID return = LOCK x_return;

INT old_link := 0;
INT suspend_link;

PROC x_get_suspend_link = INT:
BEGIN
   non_local;
   return( get_link );
   {* suspend *}
   DO
      link[ pid ] := old_link;
      old_link := 0;
      run_next   { will 'return' next time process is suspended }
   OD;
   0
END;
PROC INT get_suspend_link = LOCK x_get_suspend_link;
suspend_link := get_suspend_link;
```

```
PROC BOOL me = am_i_me;

PROC x_timer = VOID:
BEGIN
   { cp must be first non local }
   REF REF P pl;
   IF cp ISNT ( REF P(NIL))
   THEN
      IF me
      THEN
         owntime OF cp +:= pslot;
         unexpired OF cp := pslot;
         IF pid = 1
         THEN
            old_link := get_link;
            link OF cp := suspend_link
         ELSE
            link OF cp := get_link
         FI;
         REF USER u = u OF cp;
         sp OF cp := SHAKE ( tq OF u := tq OF u  APPEND cp);
         unexpired OF u -:= pslot;
         SETLINK 0;
         cp := NIL;
         SETSLOT 1000
      ELSE
         owntime OF cp +:= pslot-unexpired;
         unexpired OF cp := pslot;
         link OF cp := get_link;
         REF USER u = u OF cp;
         sp OF cp := SHAKE ( tq OF u := tq OF u  APPEND cp);
         unexpired OF u -:= pslot - unexpired;
         SETLINK 0;
         cp := NIL;
         SETSLOT 1000
      FI
   ELSE
      SETSLOT 1000
   FI
END;
PROC call_timer = INT:
BEGIN
   x_timer;
   0
END;
U call_timer; { make sure its code is loaded
                - actual call will fail }
PROC VOID timer = LOCK x_timer;


PAIR new = BREAK timer;
b OF new MAKE ref_cp;              { fill in proper ref cp }
```

```
PROC x_launch = ( PROC VOID p ) INT:
BEGIN
   non_local;
   INT return := get_link;

   PROC x_call = INT:
   BEGIN
      INT start := get_link;
      SETLINK return;
      start
   END;
   PROC INT call = LOCK x_call;

   SETLINK 0;
   call;
   p;
   link[ pid ] := 0;
   IF (num_dead +:= 1) = UPB link
   THEN
      SETLINK cont_link
   ELSE
      run_next
   FI;
   0
END;
PROC( PROC VOID ) INT launch = LOCK x_launch;


{create process shells for each of the garbage collectors}
FOR i TO n
DO
   PROC proc = VOID:
   BEGIN
      SETD 0;
      SELECTAREA 15;
      CASE U scav( i ) IN
      ( INT ok )
        (
         record(( 11, i+1 ));
         message( "GC" + intchars( i ) + " finished!" )
        ),
      ( ERRORPAIR ep )
       (
         record(( 12, i+1 ));
         IF NOT failed
         THEN
            message( "GC" + intchars( i ) + " failed" );
            failure := ep;
            failed := TRUE;
            keep_collecting := 0;
            running := FALSE
         ELSE
            message( "Now GC" + intchars( i ) + " has failed" )
         FI
       )
      ESAC
   END;
   link[ i + 1 ] := launch( LOCK proc )
OD;
```

```
{shell for the program to be tested}

SELECTAREA 14;
REF INT prog_result = HEAP INT;
SELECTAREA 15;

PROC call_prog = VOID:
BEGIN
   SETD 0;
   SELECTAREA 14;
   CASE U prog IN
   ( INT i )
     (
      record(( 11, 1 ));
      prog_result := i
     ),
   ( ERRORPAIR ep )
    (
      record(( 12, 1 ));
      IF NOT failed THEN failed := TRUE; failure := ep FI;
      keep_collecting := 0
    )
   ESAC;
   record( 6 );
   INT al;
   FOR a TO n
   DO
      al := 0;
      FOR i TO UPB table
      DO
         IF GETAREA table[ i ] = a
         THEN
            al +:= 1
         FI
      OD;
      record( ( 14, a, al ) )
   OD;
   record(( 42, num_slots ));
   keep_collecting := 0;
   running := FALSE
END;
```

```
   link[ 1 ] := launch( LOCK call_prog );

   PROC x_start_off = INT:
   BEGIN
      non_local;
      cont_link := get_link;
      pid := 0;   { first one to run will be #1 }
      run_next;
      0
   END;
   PROC INT start_off = LOCK x_start_off;

   (SYS 17) := timer;
   U start_off;    {discard any exceptions}
   (SYS 17) := old_timer;
   record(( 43, num_slots ));
   message( "Construct Stats" );

   INT ptr_stats := pb_to_d( stats[ : n_stats ] );
   IF failed THEN FAIL failure FI;
   ptr_stats  {return disc pointer to the stats}

END;

GC g_collector = LOCK x_gc

KEEP g_collector
FINISH
```

D.2 Display Statistics Program - Algol Source

The garbage collector for the experimental system for studying logical areas within a capability computer records statistics on backing store as a vector of integers. The program shown in this section interprets this data structure and produces a human readable version. Other programs, not shown, extract particular information and produce the graphs and histograms used in the main text.

```
stats_to_ed:

line_mode_m :Module
out_maker_m :Module
intchars :Module
oneline :Module
fail :Module
d_to_b :Module
concat :Module

OP ( INT ) INT                                    BSIZE = BIOP 1184;
OP ( STRUCT 3 INT ) REF VECTOR [] INT               PUN = BIOP 1001;

PROC stats_to_ed = ( INT dp ) INT: { Edfile }
BEGIN
   OUTTEXT outtext = out_maker( 60 );
   PROC(LINE)VOID out = out OF outtext; {appends lines to output}

   PROC val = ( INT num, INT wid ) LINE:
   BEGIN
      LINE str = HEAP VECTOR[ wid ] CHAR;
      FORALL c IN str DO c := " " OD;
      LINE n = intchars( num );
      str[ wid - UPB n + 1:] := n;
      str
   END;


   INT num_allocs := 1, before_allocs := 1;
                     { dont forget initial slot }
   INT local_count := 0;

   PROC output = ( INT dp ) VOID:
   BEGIN
      INT p = d_to_b( dp );
      REF VECTOR [] INT data = PUN STRUCT 3 INT( (BSIZE p)%4, p, 0 );
      IF data[ 1 ] /= 0 THEN output( data[ 1 ] ) FI;
      INT i := 2;
      WHILE i <= UPB data
      DO
```

```
INT sz = data[ i ];
i +:= 1;
INT next = sz + i;
INT type = data[ i ];
i +:= 1;
IF type = 1
THEN
   INT nb = data[ i ]; i +:= 1;
   INT nw = data[ i ]; i +:= 1;
   num_allocs +:= 1;
   out( "Allocated " + intchars( nb ) + " blocks "
                     + intchars( nw ) + " words" )
ELIF type = 2
THEN
   INT area = data[ i ]; i +:= 1;
   INT scans = data[ i ]; i +:= 1;
   out( "Area " + intchars( area ) + " scan complete. " +
        intchars( scans ) + " greys scanned." )
ELIF type = 3
THEN
   INT level = data[ i ]; i +:= 1;
   out( "Level " + intchars( level ) + " init done" )


ELIF type = 4
THEN
   INT level = data[ i ]; i +:= 1;
   out( "Level " + intchars( level ) + " scan done" )
ELIF type = 5
THEN
   INT level = data[ i ]; i +:= 1;
   out( "Level " + intchars( level ) + " recover done" )
ELIF type = 6
THEN
   out( oneline(( "------" )) );
   out( oneline(( "Computation Complete" )) );
   out( oneline(( "------" )) );
   before_allocs := num_allocs
ELIF type = 7
THEN
   INT area = data[ i ]; i +:= 1;
   INT nb = data[ i ]; i +:= 1;
   INT nw = data[ i ]; i +:= 1;
   out( "Area " + intchars( area ) + " recovered "
                + intchars( nb ) + " blocks "
                + intchars( nw ) + " words" )
```

```
ELIF type = 10
THEN
   {{  10: number of areas, map, speed, delay_time }}
   INT n = data[ i ]; i +:= 1;
   out( intchars( n ) + " areas." );
   REF VECTOR [] CHAR str;
   FOR m TO n
   DO
      str := "Area " + intchars( m ) + ": ";
      FOR j TO n
      DO
         INT l = data[ i ]; i +:= 1;
         str := str + intchars( l ) + " "
      OD;
      out( str )
   OD;
   FOR a TO n
   DO
      INT sp = data[ i ]; i +:= 1;
      out( "Speed of area " + intchars( a )
           + " is " + intchars( sp ) )
   OD;
   INT del = data[ i ]; i +:= 1;
   out( "Slice time is " + intchars( del ) );
   out( oneline(( "    " )) )


ELIF type = 11
THEN
   { 11: pid }
   INT pid = data[ i ]; i +:= 1;
   out( "Process " + intchars( pid ) + " finished" )
ELIF type = 12
THEN
   { 12: amount }
   INT num = data[ i ]; i +:= 1;
   out( intchars( num ) + " locally" );
   local_count +:= num


ELIF type = 13
THEN
   { 13: area, level, total grey scan count }
   INT area = data[ i ]; i +:= 1;
   INT level = data[ i ]; i +:= 1;
   INT count = data[ i ]; i +:= 1;
   out( "Area " + intchars( area ) + " level "
        + intchars( level )
        + " scanned " + intchars( count ) + " greys." )
ELIF type = 14
THEN
   { 14: area, num blocks alloc }
   INT area = data[ i ]; i +:= 1;
   INT alloc = data[ i ]; i +:= 1;
   out( "Area " + intchars( area ) + " has "
          + intchars( alloc )
          + " blocks still allocated" )
```

```
            ELSE
               REF VECTOR [] CHAR str := HEAP VECTOR [0] CHAR;
               FOR ind FROM i TO i + sz - 2
               DO
                  str := str + " " + intchars( data[ ind ] )
               OD;
               out( "Unknown record: " + intchars( type ) + str );
               i +:= sz - 1
            FI;
            IF i /= next THEN fail( "Format Error" ) FI
         OD
      END;


      output( dp );
      out( oneline(( intchars( before_allocs ),
                     " slots used. Then ",
                     intchars( num_allocs - before_allocs ),
                     " more."          )) );
      out( oneline(( "End." )) );
      ( end OF outtext ) ( 80 )  {return pointer to output on disc}
END

KEEP stats_to_ed
FINISH
```

D.3 Sample Statistics Output

This is a sample of the output produced by the statistics printing program. It is the complete statistics, in human readable form, produced by running the count words test on a file consisting of "hello this is a test". Note that initially no found (grey) variables are scanned because newly allocated variables are set to scanned (black).

```
2 areas.
Area 1: 1 2
Area 2: 2 1
Speed of area 1 is 50
Speed of area 2 is 50
Slice time is 16

Allocated 34 blocks 464 words
Area 1 scan complete. 0 greys scanned.
Area 2 scan complete. 0 greys scanned.
Level 2 init done
Allocated 42 blocks 607 words
Area 2 scan complete. 0 greys scanned.
Allocated 70 blocks 870 words
Area 2 scan complete. 0 greys scanned.
Process 1 finished
------
Computation Complete
------
Area 1 has 70 blocks still allocated
Area 2 has 0 blocks still allocated
Unknown record: 42 4
Allocated 105 blocks 1240 words
Area 1 scan complete. 13 greys scanned.
Level 2 scan done
Area 1 recovered 21 blocks 314 words
21 locally
Area 2 scan complete. 0 greys scanned.
Allocated 84 blocks 926 words
Area 2 level 1 scanned 0 greys.
Area 2 level 2 scanned 0 greys.
Process 3 finished
Allocated 84 blocks 926 words
Allocated 84 blocks 926 words
Allocated 84 blocks 926 words
Allocated 84 blocks 926 words
Area 1 scan complete. 32 greys scanned.
Level 2 recover done
Area 1 recovered 52 blocks 635 words
52 locally
Allocated 32 blocks 291 words
Area 1 level 1 scanned 0 greys.
Area 1 level 2 scanned 45 greys.
Process 2 finished
Unknown record: 43 4
4 slots used. Then 7 more.
End.
```

D.4 Count Words Test - Algol Source

```
count_words:

concat :Module
line_mode_m :Module
make_comp_in :Module
warning_m :Module

OP ( INT ) INT        SETAREA      = BIOP 1268;

OP ( INT ) INT        SETD     = BIOP 1194;

OP ( RES ) UNION( STRUCT( RES r ), STRUCT 2 INT )      U = BIOP 1004;
OP (STRUCT 2 INT) INT FAIL = BIOP 1273;

PROC count_words = ( INT second_area, file ) PROC INT:
BEGIN
   { count the words in the file - keep a linked list of }
   { frequency counts in another area }


   PROC words = INT:
   BEGIN

      SETAREA 1;

      MODE WORD = REF VECTOR [] CHAR,
           NODE = STRUCT( REF NODE next, WORD word, INT count );
      REF NODE end = NIL;
      REF NODE head := end;

      PROC RES reader = reader OF make_comp_input( file );
      LINE l := HEAP VECTOR [0] CHAR;
      INT next := 1;
      BOOL done := FALSE;
```

```
PROC next_word = WORD:
BEGIN
   WHILE next > UPB l
   ANDTH NOT done
   DO
      CASE U reader IN
      ( STRUCT( RES r ) r )
       (
         CASE r OF r IN
         ( LINE line )
          (
            l := line;
            next := 1;
            WHILE next <= UPB l
            ANDTH l[ next ] = " "
            DO
               next +:= 1
            OD
          )
         OUT
            SKIP
         ESAC
       )
      OUT
         done := TRUE
      ESAC
   OD;


   IF done
   THEN
      HEAP VECTOR [0] CHAR
   ELSE
      INT start = next;
      INT i := next + 1;
      WHILE i <= UPB l
      ANDTH l[ i ] /= " "
      DO
         i +:= 1
      OD;
      next := i;
      WHILE next <= UPB l
      ANDTH l[ next ] = " "
      DO
         next +:= 1
      OD;
      l[ start : i-1 ]
   FI
END;
```

```
      PROC find = ( WORD w, REF REF NODE head ) BOOL:
      BEGIN
         REF NODE p := head;
         BOOL found := FALSE;
         WHILE ( p ISNT end )
         ANDTH NOT found
         DO
            IF w = word OF p
            THEN
               found := TRUE;
               count OF p +:= 1
            ELSE
               p := next OF p
            FI
         OD;
         found
      END;


      REF VECTOR [] CHAR word;
      WHILE
         word := next_word;
         UPB word > 0
      DO
         SETAREA second_area;
         IF NOT find( word, head )
         THEN
            head := HEAP NODE := ( head, word, 1 )
         FI;
         SETAREA 1
      OD;
      0
   END;

   words

END

KEEP count_words
FINISH
```

D.5 Count Words Test - Test Input

Here is the sample text used as input to the count words test. It is a fragment of the on-line tutorial information for Flex.

```
                    Simple editing operations

This section is to introduce you to typing new text and making
simple alterations to it. It concerns text which is organised in
individual lines rather than English text. English text is organised
in paragraphs. The editor does handle paragraphs, but they are
slightly more complex and will be described later.

Read the next paragraph through before doing a double edit.

If you press the Double Edit key (Acc on the Perq keyboard, towards
the top left) you will get two windows. Make sure that the puck is
inside the window before pressing Double Edit. The upper or
left-hand one will contain the sub-text that you are already
editing, and the paper may have been moved to keep the cursor in the
new smaller window. The lower or right-hand one will contain a clean
sheet of paper. You can move from one window to another by using the
puck. If, while the cursor is in one window, you press the select
button on the puck and try to move it out of the window while
holding it down, then the paper will move behind the window in the
usual way. But if you move the puck into the other window before
pressing the select button, the cursor will follow into the other
window. Leave the cursor on the asterisk below, and now press Double
Edit.

*


You can return from the double edit back to normal editing by
typing CTRL OOPS while the cursor is in the top half. Move
between the two windows, type a bit in the lower one, and go back to
single edit. What you type in the lower half is discarded.


The keys in the right hand key-pad and some of the keys around the
edge of the keyboard, are used to perform editing functions. You may
have some way of re-labelling these keys, or you may have a printed
diagram of them. If not, the next page contains an annotated
diagram. You can move onto the next page by pressing the Next Page
key (9 on the keypad) and back again by pressing the Previous Page
key (8 on the keypad).


The key labelled south-west (called "oops" on the Perq keyboard) is
another cursor movement key. It moves down a line and to the left
end. Back-space removes the character before the cursor and steps
back one place.


You cannot alter this file: if you try to there will be no effect but
a beep from the sound generator. So to try the keys which are
described below you will have to use Double Edit.
```

Remember which keys are used for Delete Character, Delete Element, Duplicate, Insert Blank, Insert Remembered Vertical and Insert Below.

Delete Character removes the character on which the cursor is resting. Delete Element will remove the line in which the cursor lies. Insert Blank Line will put a new blank line in. Insert Below puts in a blank line below the line the cursor is on and moves the cursor down on to it in a position immediately below the first character of the current line. This is useful for typing at the bottom of the paper.

Since you cannot alter this text, in order to do experiments you will have to do a double edit and use the lower window. Go into the second window, type some lines and use the editing keys described above.

If you type more characters than will fit onto the line you will hear a beep, and the character is not inserted. There is no automatic roll-over to the next line, since it is assumed that because this is text in which the layout is important, you will want to make the necessary adjustments. There is roll-over if you are typing paragraphs, since this is appropriate for English text. There are facilities to be described later which will help in reorganising a non-paragraph line if you find it will not fit in.

When you deleted a line the left hand end of the very top line of the screen changed. It tells you the number of elements which are being remembered. When you delete something it is remembered (last-in, first-out). The Insert Remembered Vertical key puts back the top remembered element (the last one deleted) where the cursor is. So if you accidentally deleted something and you want to get it back, this is how you can do it. It is also useful for moving a line or lines from one place to another. If you delete some lines, move the cursor somewhere else and put them back, you have done a simple cut-and-paste operation. The Duplicate key puts items into the remembered elements without deleting them. You can use this to copy sections from one place to another. Experiment with these keys in the second window. You can also use the Duplicate key to copy lines from the first window into the second. You can't use Delete for this purpose, because that would mean altering the tutorial, which has been prevented, but in ordinary situations where alteration is not prohibited you could use this key.

Double Edit is mainly used to consult information while keeping sight of something, or to transfer information from one text to another while keeping sight of both source and destination, or for providing a work space so that we can try something out without spoiling the appearance of the main text. The last purpose is what we shall mostly use it for in the tutorial, because the tutorial text is protected against changes, but don't forget the other uses.

If you don't want the screen to be split in two with a horizontal line when you press Double Edit, you can control it with the puck. Point the puck just above the window in which the cursor lies and press Double edit. The window is split with a vertical line at the point near the puck. Similarly if you point just to the left of the current window. If you split it in the wrong place you can immediately undo it by using CTRL OOPS.

Résumé

You have

1) Typed characters
2) Deleted characters
3) Deleted lines
4) Recovered lines after accidental deletion
5) Moved a group of lines from one place to another
6) Copied lines from one place to another
7) Inserted a blank line to type on
8) Inserted a blank line below the current one
9) Done a double edit
10) Moved text between windows

When you type CTRL OOPS in the upper or left-hand window you will go back to single edit.

The next page is the diagram of the keyboard. The page after that is a continuation of the tutorial.

To read the next page, press the Next Page key (9 on the keypad).

D.6 Routing Test - Algol Source

```
routing:

maxmin_m :Module
make_comp_in :Module
concat :Module
oneline :Module
intchars :Module
warning_m :Module
roll_m :Module

OP ( INT ) INT        SETAREA      = BIOP 1268;

MODE RVC = REF VECTOR [] CHAR,
     RVI = REF VECTOR [] INT,
     MSG = STRUCT( RVC text, RVI route, return, BOOL ack ),
     NODE = STRUCT( REF NODE next, other, REF MSG msg ),
     SITE = STRUCT( REF NODE in, out );


REF NODE nil = NIL;

PROC number = ( RVC line, REF INT p ) INT:
BEGIN
   INT res := 0;
   WHILE p <= UPB line ANDTH line[ p ] /= " "
   DO
      res := res * 10 + ABS line[ p ] - ABS "0";
      p +:= 1
   OD;
   res
END;


OP + = ( REF VECTOR [] INT a, b ) REF VECTOR [] INT:
BEGIN
   HEAP VECTOR [ UPB a + UPB b ] INT new;
   new[ : UPB a ] := a;
   new[ UPB a + 1 : ] := b;
   new
END;


PROC append = ( REF NODE node, REF REF NODE head ) VOID:
BEGIN
   REF REF NODE p := head;
   WHILE p ISNT nil
   DO
      p := next OF p
   OD;
   REF REF NODE( p ) := node
END;
```

```
PROC remove = ( REF NODE node, REF REF NODE head ) VOID:
BEGIN
   REF REF NODE p := head;
   WHILE other OF p ISNT node
   DO
      p := next OF p
   OD;
   REF REF NODE( p ) := next OF p
END;


PROC route = ( INT edfile, VECTOR [] INT area ) PROC INT:
BEGIN

   VECTOR [ UPB area ] SITE site;
      FORALL s IN site DO s := ( nil, nil ) OD;

   INT num_msgs := 0;
   INT next_site := 0;

   PROC transfer = ( REF MSG m ) VOID:
   BEGIN
      return OF m := (HEAP VECTOR [1] INT := (route OF m)[ 1 ] )
                                              + return OF m;
      INT n = UPB route OF m - 1;
      route OF m := HEAP VECTOR [ n ] INT := (route OF m)[ 2 : ]
   END;


   PROC enter_message = ( MSG msg ) VOID:
   BEGIN
      HEAP MSG m := msg;
      INT from = (return OF m)[ 1 ],
          first = (route OF m)[ 1 ];
      HEAP NODE in := ( nil, nil, m );
      HEAP NODE out := ( nil, nil, m );
      other OF in := out;
      other OF out := in;
      append( in, in OF site[ first ] );
      append( out, out OF site[ from ] );

      roll( oneline(( text OF m, " from ", intchars( from ),
                                  " to ", intchars( first )    )) );
      warning( oneline(( text OF m, " from ", intchars( from ),
                                  " to ", intchars( first )    )) )
   END;
```

```
PROC move_message = VOID:
BEGIN
   FOR s TO UPB site WHILE SETAREA area[ s ]; TRUE
   DO
      REF NODE p := in OF site[ s ];
      roll( oneline(( "Site ", intchars( s ), ":" )) );
      WHILE p ISNT nil
      DO
         roll( oneline(( "   from ",
                         intchars( (return OF msg OF p )[ 1 ] ),
                         ": ", text OF msg OF p )) );
         warning( oneline(( "   from ",
                            intchars( (return OF msg OF p )[ 1 ] ),
                            ": ", text OF msg OF p )) );
         p := next OF p
      OD;
      roll( "...." );
      warning( "...." )
   OD;


   WHILE
      next_site +:= 1;
      IF next_site > UPB area THEN next_site := 1 FI;
      in OF site[ next_site ] IS nil
   DO
      SKIP
   OD;


   SETAREA area[ next_site ];

   REF MSG msg = msg OF in OF site[ next_site ];
   INT from = (return OF msg)[ 1 ];
   remove( in OF site[ next_site ], out OF site[ from ] );
   in OF site[ next_site ] := next OF in OF site[ next_site ];

   SETAREA area[ (route OF msg)[ 1 ] ];


   IF UPB route OF msg = 1
   THEN
      IF ack OF msg
      THEN
         roll( oneline(( text OF msg, " acked" )) );
         warning( oneline(( text OF msg, " acked" )) );
         num_msgs -:= 1
      ELSE
         roll( oneline(( text OF msg, " arrived" )) );
         warning( oneline(( text OF msg, " arrived" )) );
         RVI r = route OF msg;
         route OF msg := return OF msg;
         return OF msg := r;
         ack OF msg := TRUE;
         enter_message( msg )
      FI
   ELSE
      transfer( msg );
      enter_message( msg )
   FI
END;
```

```
   PROC RES next = reader OF make_comp_input( edfile );

   PROC sim = INT:
   BEGIN
      BOOL running := TRUE;
      INT delay := 0, p;
      WHILE running
      DO
         TO delay WHILE num_msgs > 0
         DO
            move_message
         OD;
         CASE next IN
         ( RVC line )
          (
            { each line is either a delay number or
              some text (ending with ":") and
              some numbers separated by
              spaces (eg. "hello:1 2 3") or "end" }
            IF "0" <= line[ 1 ] ANDTH line[ 1 ] <= "9"
            THEN
               delay := number( line, p := 1 )
            ELIF line = "end"
            THEN
               running := FALSE


            ELSE
               p := 1;
               WHILE line[ p ] /= ":"
               DO
                  p +:= 1
               OD;
               RVC text = HEAP VECTOR [ p ] CHAR := line[ : p ];
               INT first = number( line, p +:= 1 );

               SETAREA area[ first ];

               REF VECTOR [] INT route := HEAP VECTOR[1]INT := first;
               WHILE p <= UPB line
               DO
                  route := route +
                            (HEAP VECTOR [1] INT
                               := number( line, p +:= 1 ))
               OD;
               MSG msg := (text, route, HEAP VECTOR [0] INT, FALSE);
               transfer( msg );
               enter_message( msg );
               num_msgs +:= 1;
               delay := 1
            FI
          )
         ESAC
      OD;
      0
   END;
   sim
END

KEEP route
FINISH
```

D.7 Distributed Capability System Experiment - Algol Source

This is the source text of the statistics gathering software inserted into the mechanism that triggers local garbage collection and into the remote capability garbage collector.

```
stats:

remote_mode :Module
beep_m :Module
warning_m :Module
concat :Module
intchars :Module
oneline :Module

REF VECTOR [] INT st := HEAP VECTOR [1000] INT;
INT p := 0;
BOOL overflow := FALSE;
BOOL enabled := FALSE;
INT initial_time := 0;

INT panic = UPB st - 10;

{ Record: Type (Local/Global),
          Number Remote Caps deleted,
          wanted store + global recovered + local recovered
          wanted store + global recovered
          wanted store
          start time
          end time
}


MODE STATS = PROC( REF REMOTE, REF VECTOR [] INT,
                               REF VECTOR [] INT )VOID;
MODE BUFF = REF VECTOR [] INT;
MODE PAIR = STRUCT( INT i,j );

OP ( STATS ) STATS                                  LOCK = BIOP 1259;
OP ( PROC(INT)VOID ) PROC(INT)VOID                  LOCK = BIOP 1259;
OP ( PROC VOID ) PROC VOID                          LOCK = BIOP 1259;
OP ( PROC INT ) PROC INT                            LOCK = BIOP 1259;
OP ( PROC PROC BUFF ) PROC PROC BUFF                LOCK = BIOP 1259;
OP ( PROC(REF BUFF)VOID ) PROC(REF BUFF)VOID        LOCK = BIOP 1259;

OP ( INT ) INT                                  GEN = BIOP 1173;
OP ( INT ) INT                              ADDRESS = BIOP 1210;
OP ( INT ) REF PROC INT                      REFSYS = BIOP 1317;
OP ( INT ) REF INT                              SBI = BIOP 1317;
OP ( PAIR ) INT                             EXITFAIL = BIOP 1169;
OP ( INT ) VOID                              SETSLOT = BIOP 1316;


INT                                         scavenge = BIOP 1204;
STRUCT 5 INT                                   dumpu = BIOP 5206;
INT                                          resetu = BIOP 1207;
INT                                        unexpired = BIOP 1036;
INT                                         time_now = BIOP 1035;
```

```
PROC 1_globals = ( REF REMOTE r,
                   REF VECTOR [] INT before, after ) VOID:
BEGIN
   IF enabled
   THEN
      INT start = time_now;
      INT local = scavenge;
      FORALL b IN before DO b := 0 OD;
      INT global = scavenge;
      INT wanted = ABS( ( BIN ADDRESS GEN 0 ) AND 16r7ffff );

      IF p > panic THEN overflow := TRUE; p := 0 FI;

      st[ p +:= 1 ] := 1;
      st[ p +:= 1 ] := UPB before - UPB after;
      st[ p +:= 1 ] := wanted + local + global;
      st[ p +:= 1 ] := wanted + local;
      st[ p +:= 1 ] := wanted;
      st[ p +:= 1 ] := start - initial_time;
      st[ p +:= 1 ] := time_now - initial_time;
      beep
   FI
END;


STATS globals = LOCK 1_globals;
PROC INT g;

PROC 1_start_local = INT:
BEGIN
   g := REFSYS 4;
   scavenge;
   PROC 1_new_gc = INT:
   BEGIN
      STRUCT 5 INT du = dumpu;

      INT start = time_now;
      INT slottimeleft = unexpired;
      SETSLOT 10000;
      (SBI 1):=0;


      INT recovered = scavenge;
      INT wanted = ABS( ( BIN ADDRESS GEN 0 ) AND 16r7ffff );

      IF p > panic THEN overflow := TRUE; p := 0 FI;

      st[ p +:= 1 ] := 2;
      st[ p +:= 1 ] := 0;
      st[ p +:= 1 ] := wanted + recovered;
      st[ p +:= 1 ] := wanted + recovered;
      st[ p +:= 1 ] := wanted;
      st[ p +:= 1 ] := start - initial_time;
      st[ p +:= 1 ] := time_now - initial_time;

      SETSLOT unexpired;
      IF recovered <= SBI 2 THEN resetu; EXITFAIL PAIR (-1,-1) FI;
      resetu
   END;
   (REFSYS 4) := LOCK 1_new_gc;
   (SBI 2) := -1;
   (REFSYS 4);  { load code block }
   0
END;
PROC INT start_local = LOCK 1_start_local;
```

```
PROC l_stop_recording = VOID:
BEGIN
   enabled := FALSE;
   (REFSYS 4) := g
END;
PROC VOID stop_recording = LOCK l_stop_recording;


PROC l_start_recording = PROC REF VECTOR [] INT:
BEGIN
   initial_time := time_now;

   PROC l_get_stats = ( REF REF VECTOR [] INT buffer ) VOID:
   BEGIN
      buffer := buffer[ : p ] := st[ : p ];
      p := 0
   END;
   PROC( REF REF VECTOR [] INT ) VOID get_stats = LOCK l_get_stats;


   HEAP VECTOR [0] INT no_ints;

   PROC record = REF VECTOR [] INT:
   BEGIN
      IF p /= 0
      THEN
         REF VECTOR [] INT buffer := HEAP VECTOR [ UPB st ] INT;
         get_stats( buffer );
         buffer
      ELSE
         no_ints
      FI
   END;
   enabled := TRUE;
   p := 0;
   start_local;
   record
END;

PROC PROC REF VECTOR [] INT start_recording = LOCK l_start_recording

KEEP globals, start_recording, stop_recording
FINISH
```